

AFRL-IF-RS-TR-2003-2
Final Technical Report
January 2003



PROGRAMMING HIGH PERFORMANCE RECONFIGURABLE COMPUTERS (HPRC)

CACI Technologies, Incorporated

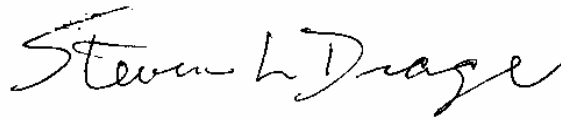
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-2 has been reviewed and is approved for publication.

APPROVED:

A handwritten signature in black ink, reading "Steven L. Drager". The signature is written in a cursive style with a large, stylized 'S' and 'D'.

STEVEN L. DRAGER
Project Engineer

FOR THE DIRECTOR:

A handwritten signature in black ink, reading "Michael L. Talbert". The signature is written in a cursive style with a large, stylized 'M' and 'T'.

MICHAEL L. TALBERT, Maj., USAF
Technical Advisor, Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE JANUARY 2003	3. REPORT TYPE AND DATES COVERED Final Apr 01 – Jul 02	
4. TITLE AND SUBTITLE PROGRAMMING HIGH PERFORMANCE RECONFIGURABLE COMPUTERS (HPRC)			5. FUNDING NUMBERS C - F30602-00-D-0221/0005 PE - 62702F PR - 459T TA - QF WU - 06	
6. AUTHOR(S) Gregory D. Peterson				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Prime: CACI Technologies, Incorporated 14151 Park Meadow Drive Chantilly Virginia 20151 SUB: University of Tennessee Electrical and Computer Engineering Knoxville Tennessee 37996			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTC 26 Electronic Parkway Rome New York 13441-4514			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-2	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Steven L. Drager/IFTC/(315) 330-2735/ Steven.Drager@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) The integration of High Performance Computing with Reconfigurable Computing offers great potential for increased performance and flexibility for a wide range of computing problems. High Performance Computing architectures and Reconfigurable Computing systems have independently demonstrated performance advantages for applications such as digital signal processing and pattern recognition. By exploiting the near hardware specific speed of Reconfigurable Computing systems incorporated into a computer cluster, there is potential for significant performance advantages over software-only or uniprocessor solutions. However, application development barriers exist that will slow the widespread adoption of this technology. This report presents the results of research seeking to overcome one of these barriers, the development of a programming framework for High Performance Reconfigurable Computing systems.				
14. SUBJECT TERMS High Performance Computing, Reconfigurable Computing, Programming Environments			15. NUMBER OF PAGES 76	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

Introduction.....	1
Reconfigurable Computing.....	2
High Performance Reconfigurable Computing	3
Related Work	5
Programming Environments	6
Models of computation in Ptolemy II	7
Parallel Programming	8
Reconfigurable Computing Development	9
Mathematical Foundations.....	13
Overview.....	15
Reconfigurable Computing Node Analysis	17
High Performance Reconfigurable Computing Multi-Node Analysis.....	19
High Performance Reconfigurable Computing Development Hardware	22
Reconfigurable Computing Model	22
High Performance Reconfigurable Computing Model	25
Model Applications.....	27
HPRC Application Examples	29
Image Segmentation.....	31
Conclusions and Future Work	32
Acknowledgements.....	33
References.....	34
Appendix A: Firebird Host Source Code.....	38
Appendix B: MATLAB Makefile Code	61
How To Call Matlab Functions from C Programs:.....	66

Table of Figures

Figure 1. High Performance Reconfigurable Computer (HPRC) Architecture	2
Figure 2. Synchronous Iterative Algorithm	17
Figure 3. Flow of Synchronous Iterative Algorithm for Multi Node Analysis	20
Figure 4. Speedup Curves: a) Vary number of RC units, b) Vary configuration time, and c) Vary total work	24
Figure 5. Comparison of RC Model Prediction with Measurement Results	25
Figure 6. Speedup Curves: a) Vary work for one FPGA per RC unit and b) Vary work for two FPGAs per RC unit and c) Increasing number of nodes (one RC unit per node) work fixed	26
Figure 7. SoC Architecture Example	28

Table of Tables

Table 1. Symbols and Definitions.....	16
Table 2. Model Parameters for Wildforce from Benchmark Application	23
Table 3. Runtime Predictions and Measurements (time in seconds)	25

Introduction

A number of important defense and industrial applications present computational demands greatly exceeding the capabilities of consumer (desktop or mainframe) computers. For decades, these specialized demands have driven the development and evolution of a supercomputing industry. Moore's Law predicts the doubling of semiconductor performance every eighteen months. Because it has held true for nearly four decades (and most experts expect it to hold for at least the next 10-15 years), the capabilities of high performance computing enjoy the revolutionary impact of continued exponential growth. At the same time that processors have followed Moore's Law, the computational power of configurable logic has grown even faster. In less than two decades, FPGAs (Field Programmable Gate Arrays) have grown from small prototyping or "glue logic" applications to the point of now holding nearly ten million gates of logic. The impressive capacity and much-improved speed of FPGAs have made them useful in many applications, including some with significant computational demand. Although the combination of supercomputing platforms with configurable logic seems like an attractive approach for achieving tremendous performance, many challenges exist as well. The most appropriate architectures, computational models, runtime systems, applications, and development environments must be considered. This technical report discusses research aimed at developing an initial programming infrastructure for this emerging supercomputing technology.

High Performance Computing (HPC) is the use of multiple processors or processing nodes collectively on a common problem. Reconfigurable Computing (RC) is the combination of reconfigurable logic with a general-purpose microprocessor. The architectural intent is to achieve higher performance than typically available from software-only solutions with more flexibility than achievable with hardware ASICs. In RC architectures, the microprocessor performs those operations that cannot be done efficiently in the reconfigurable logic such as loops, branches, and possible memory accesses, while computational cores are mapped to the reconfigurable hardware [34].

High Performance Reconfigurable Computing (HPRC) is the architectural combination of High Performance Computing and Reconfigurable Computing. The proposed HPRC platform as shown in Figure 1 consists of a number of distributed computing nodes connected by some interconnection network (the network can be a switch, hypercube, systolic array, etc. using Ethernet, Myrinet, or some other networking technology), with some or all of the computing nodes having RC element(s) associated with them. The HPRC platform will potentially allow users to exploit the performance speedups commonly achieved in parallel systems in addition to the speedup offered by reconfigurable hardware coprocessors. Many applications such as image or signal processing algorithms and various simulation algorithms stand to benefit from the potential performance of this architecture.

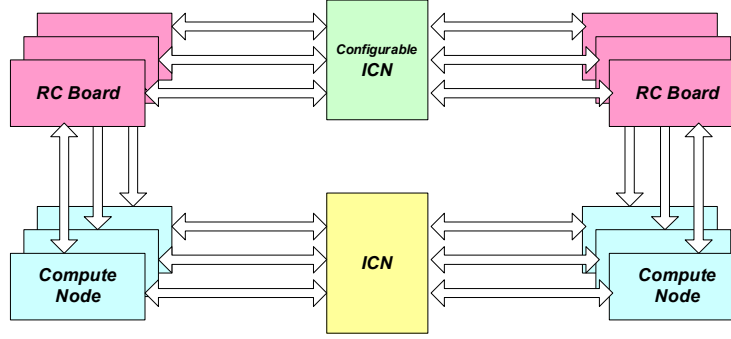


Figure 1. High Performance Reconfigurable Computer (HPRC) Architecture

The addition of a configurable network connecting the RC elements offers even more performance advantages for many applications such as discrete event simulation and many numeric computations. The additional network would provide a less inhibited route for synchronization, data exchange, and other communications between processing nodes. Research by Chamberlain [4,5,6], Reynolds et. al. [38,39,40], and Underwood et. al. [44] all confirm the performance benefits of a specialized configurable network for applications with barrier synchronization events or applications requiring the exchange of large amounts of data.

As individual platforms for computing, HPC and RC are challenging enough to program and utilize effectively. Combining these powerful domains will require the development of new analysis and design tools. In this report, we discuss efforts to develop a performance modeling infrastructure for HPRC, model communications between the various reconfigurable computing and processing elements, evaluate the most appropriate computation models for HPRC, and consider the programming effectiveness of different languages, tools, and libraries for HPRC.

A performance modeling framework with models describing this new architecture will help not only in understanding and exploiting the design space but will be a building block for many of these tools. The performance of the HPRC system is affected by architectural variables such as number of nodes, number of FPGAs, FPGA size, heterogeneity, and network performance, just to name a few, and the available permutations make the design space extremely large. Without a modeling framework to assist with the analysis of these issues, tradeoffs cannot be effectively analyzed, potentially resulting in grossly inefficient use of the resources.

Reconfigurable Computing

Reconfigurable computing (RC) is the coupling of reconfigurable hardware such as FPGAs to general-purpose processors. Some examples of the potential coupling of the reconfigurable hardware are a PCI bus card with FPGA or PLD (Programmable Logic Device) devices, a reconfigurable co-processor mapped in the processors memory space, or an area of reconfigurable logic integrated on the same die with the processor. Research has shown that many of today's computationally intensive applications can

benefit from the speed offered by application specific hardware co-processors. For applications with multiple specialized needs, it is not feasible to have a different co-processor for every specialized function. Such diverse applications stand to benefit the most from the flexibility of RC architectures since one RC unit can provide the functionality of several ASIC co-processors in a single device. Several research groups have demonstrated successful RC architectures [12,13,15,16,18,25,27,28,44,46,48].

There are many RC systems available from companies such as Annapolis Microsystems [2], Nallatech [29], Virtual Computer Corporation [45], and research organizations such as University of Southern California's Information Sciences Institute (ISI) [18], The Chinese University of Hong Kong [25], and Carnegie Mellon University [13,28]. The Wildforce and Firebird units from Annapolis Microsystems [2] and the SLAAC units from ISI [18] are all PCI-bus cards with onboard memory. The Pilchard architecture developed by The Chinese University of Hong Kong [25] interfaces through the memory bus for closer coupling with the processor. The PipeRench reconfigurable fabric [13,28] is an interconnected network of configurable logic and storage elements that uses pipeline reconfiguration to reduce overhead which is one of the primary sources of inefficiency in other RC systems.

High Performance Reconfigurable Computing

The proposed HPRC platform consists of a system of RC nodes connected by some interconnection network (switch, hypercube, array, etc.). Each of the RC nodes may have one or more reconfigurable units associated with it. This architecture provides the user with the potential for more computational performance than traditional parallel computers or reconfigurable coprocessor systems alone.

The HPRC architecture offers many architecture options. Starting with the traditional HPC aspects of the HPRC architecture, there are many network and processor considerations. These options alone make performance analysis complicated and interesting. Adding in the options available for RC such as the coupling of reconfigurable hardware to the processor, number of reconfigurable units, size of FPGA(s), dedicated interconnection network, and others, the analysis problem becomes enormous. Understanding these issues and how they affect the overall system performance is vital in exploiting this potentially powerful architecture. An analytical modeling framework addressing these issues will be a key tool towards understanding through analysis.

HPRC could prove a very useful platform for applications involving heavy number crunching and rigorous computations. Examples of these would be signal and image processing algorithms. It can also be used to perform time-consuming simulations – it is natural to expect them to improve simulation times considerably given that they use two speed-up techniques simultaneously. A number of algorithms to solve graph problems also seem promising for HPRC applications.

Parallel supercomputers have moved from vector supercomputing technologies to microprocessor-based massively parallel computers. With the promulgation of both powerful computers and fast communications technologies, a huge amount of potential computational capabilities exists. Grid-based computing technologies seek to exploit these resources, particularly the cycles that would otherwise be lost for idle computers. Communications libraries, distributed job schedulers, and related software developments combined with the availability of powerful, inexpensive computational and communications infrastructure has lead to the popularity of Beowolf clusters. Using a number of clustered commodity PCs, an organization can create a powerful parallel computing infrastructure. The primary difference between Beowolf clusters and high performance computers is the optimized interconnection networks used by HPC platforms.

Although FPGA boards are not as inexpensive (yet), one can assemble a powerful HPRC platform for a modest hardware investment. In this report, we discuss several applications that illustrate the potential for HPRC applications to speed up computations. For a number of important defense applications, such as in signal and image processing, there is significant potential for HPRC to help accelerate these critical applications.

One possible criticism of the HPRC approach for defense applications, especially for signal and image processing, is that a Beowolf cluster or HPC platform augmented with reconfigurable computing elements may have limited potential for packaging in embedded systems. Hence, aerospace or man-portable use of this technology may not be practical.

As Moore's Law continues its inexorable pace, the number of transistors available for use on integrated circuits continues to grow exponentially. By the end of the decade, chips with over a billion gates will be available. With this tremendous growth, the question of how these transistors can be exploited is a concern. Even now, design engineer productivity is severely lagging behind current demands. To address this, large circuits will include a collection of processors, cores for specific functions, and reconfigurable logic. In short, HPRC platforms will be available on a chip. Hence, this effort not only addresses programming HPRC platforms one would expect to find in a machine room or equipment rack, it also addresses critical needs in application development for next-generation embedded systems.

This trend towards the convergence of parallel processing and reconfigurable computing is already evident. Reconfigurable logic device manufacturers such as Xilinx, Altera, Atmel, Cypress, and others are already bringing together their reconfigurable logic devices with embedded microprocessors and microcontrollers. In the case of the Xilinx Virtex II Pro family, multiple PowerPC cores are integrated on the same die as tens of millions of gates of FPGA circuitry. In addition, a number of high-speed communications links are provided to get data into and out of the parts. HPRC on a chip is not only technically feasible it is currently commercially available. The problem of effectively programming these platforms remains.

Related Work

A significant amount of related work has been performed under the auspices of the DARPA Adaptive Computing Systems (ACS) program. The most relevant work under the ACS program addresses the application development task. The Synopsys Nimble Compiler effort targets the synthesis of C onto reconfigurable logic, but is not oriented for high performance computing tasks. The USC/ISI DEFACTO effort exploits the SUIF project to address compilation of more abstract representations onto reconfigurable hardware, and includes some nice capabilities to map to customized ALU/CPU structures for a given program. Once again, this is a very low-level view of the partitioning and mapping problem. The compilation technology from these programs can be applied to compiling function blocks to reconfigurable computing elements once partitioning is completed and the blocks are mapped to RC elements.

Efforts from Northwestern University and UT exploit visual programming languages like Khoros or MATLAB to ease the reconfigurable hardware programming task. These approaches provide an excellent programming model for applications developers. These projects do not currently support the migration of functionality between CPUs and RC elements in either a static or dynamic manner. This effort addresses static mapping between CPUs and RC elements in a HPRC architecture, while considering the evaluation capabilities required for a dynamic mapping capability.

There are several Java-related ACS programs for applications targeting FPGAs. Brigham Young University researchers have developed the JHDL Java-based hardware description language environment for programming FPGAs. The approach enjoys the benefits of code portability inherent with Java, while still achieving good hardware performance on the FPGA devices. With respect to hardware/software systems, the approach suffers from the reduced performance typically encountered with Java interpreters or compilers, which limits its applicability to high performance computing environments. A related effort from Lava Logic (a division of TSI Telesys) targets the use of Java for programming reconfigurable devices by developing hardware implementations for Java bytecodes, but it suffers from the same high performance computing software drawbacks as the JHDL effort. The Xilinx JBITS effort targets the use of the fine-grain reconfigurability of the Virtex family of FPGA devices. This ability to perform fine-grain reconfigurability during operation promises a performance boost, but focuses on low-level design issues. Hence, we can easily leverage such advances, but the work discussed here addresses a different programming problem.

The System Level Applications of Adaptive Computing (SLAAC) program provides the primary demonstration vehicle for the DARPA ACS program. The SLAAC RC architecture has been developed and used for a number of technology demonstrations. In addition SLAAC completed compilation work to map C to VHDL on Annapolis Microsystems RC boards. The Virginia Tech ACS work uses the BYU JHDL design environment, but it does not have automatic partitioning onto multiple FPGAs. The manual intervention required for this task is significant; supporting multiple RC boards further complicates the design task. The SLAAC API is being developed at Virginia

Tech to make porting application host code between reconfigurable computing boards easier.

In an attempt to better support portability of vector, signal, and image processing applications, the embedded systems community created the VSIPL API for commonly used functions. Vendors for various HPC platforms support the VSIPL API by providing a library of functions optimized for their particular machines. Because a variety of vendors support VSIPL, applications developers can develop their applications using familiar programming languages/tools like C/C++, MATLAB, or Khoros, and can more easily port their applications to other platforms. Unfortunately, the VSIPL library does not currently support parallel processing, although preliminary work has begun on developing a parallel VSIPL extension. A reasonable strategy for making effective use of HPRC platforms would exploit the parallel VSIPL infrastructure for HPC platforms to provide hardware/software codesign support for automatic partitioning and mapping of tasks between multiple processors and multiple FPGAs. The proposed research takes important steps in this direction and helps validate the appropriateness of the approach.

Programming Environments

It would be ideal to have a tool that would let users design their application and map it onto the HPRC system without having to know the details of parallel processing or reconfigurable computing. Such a tool would essentially achieve the following:

User → CAD Tool → design specification → HPRC

In this research, we are investigating the efficacy of using the Ptolemy II framework [21,22,37], MATLAB, Khoros, or other popular tools to program HPRC platforms. We know that the modeling framework for the HPRC architecture is also suitable to the design of Systems on a Chip (SoC), because the design and architecture of HPRC and SoCs will be homomorphic. Both the architectures contain Reconfigurable Units (RC units) and also share some of the same design methodologies in Hardware/Software Codesign [23].

Previous work at UT focused on the use of Khoros, a graphical design environment primarily used for image processing applications. The CHAMPION program used Khoros to capture algorithms, and then provided a set of VHDL libraries to map the application to a reconfigurable computing board with one or more FPGAs. A partitioning algorithm automatically broke the application into pieces that were then mapped to the collection of available FPGAs. Using this approach, speedups of up to 2000x were achieved in mapping applications to FPGAs.

MATLAB is another tool for capturing algorithms and is widely used. We discuss the interfaces provided with MATLAB for using parallel processing and how we can develop the interface between a processor running a MATLAB program and the FPGAs we would like to use to accelerate portions of the MATLAB problem. First, we discuss

some issues related to computational models and how we are investigating the models of computation one may wish to consider for HPRC applications.

Models of computation in Ptolemy II

Ptolemy is a software framework developed at the University of California, Berkeley and is used for modeling, simulation and design of concurrent, real-time embedded systems. The advantage of this tool is that it allows heterogeneous mixing of different “models of computation”. A model of computation varies from another mainly in its notion of “time”. Ptolemy II is a JAVA-based component-assembly framework and its GUI allows the user to create designs that involve one or more interacting components (which could be of different models of computation) [24].

“Models of computation” are architectural patterns, which focus on relationships between concurrent or sequential components. Ptolemy II includes a suite of *domains*, each of which realizes a model of computation. It also includes a component library, in which most components are *domain polymorphic*, in that they can operate in several of the domains. Most are also *data polymorphic*, in that they operate on several data types. The domains that have been implemented are listed below.

CT: continuous-time modeling,

DE: discrete-event modeling,

FSM: finite state machines,

PN: process networks,

SDF: synchronous dataflow

CSP: communicating sequential processes, (only in the full release)

DDE: distributed discrete events (experimental),

DT: discrete time, (experimental),

Giotto: periodic time-driven (experimental) and

GR: 3-D graphics (experimental).

SR: Synchronous Reactive (experimental).

TM: Timed Multitasking (experimental).

The first aim of the research into computational models is to demonstrate the feasibility of using HPRC for the applications/computational models chosen. For each computational model, the process of dividing tasks between the processors and the reconfigurable units is done by hand. In so doing, a better understanding of the types of applications and the implementation approaches that are most appropriate for HPRC architecture. This insight is critical to the ultimate goal is to find a way to efficiently automate the process while achieving high performance. The applications we consider for HPRC include different computational models, so these demonstration applications help in understanding which computational models are most appropriate for HPRC

platforms. Although a sampling of results has been obtained using Ptolemy II and other tools for HPRC applications, this work is still preliminary and in progress. Some comments are included in the application discussion below concerning their computational model and the effectiveness in mapping the application/computational model to HPRC.

Parallel Programming

The Grid is the name given for the vast collection of interconnected computers distributed throughout the world, combined with the software tools and infrastructure required to develop and execute applications on these computational resources. A wide variety of tools provide support for job scheduling, including such tools as Condor, Utopia, and LSF. Other tools such as the Internet Backplane Protocol for Logistical Computing help in caching data for distributed grid applications. Distributed operating system work, including the Globus project, seek to exploit grid resources as well by allowing tasks to be spawned off to other machines and migrated as necessary.

The programming infrastructure to support communicating processes executing on the grid is a quite difficult problem. Early efforts to provide these capabilities include PVM and MPI, which are libraries of communications and control functions to coordinate processes on the grid. PVM is quite popular and widely distributed, with support for a wide variety of architectures. It supports a dynamic process model; hence tasks can be created or destroyed at runtime. In contrast, MPI supports a CSP-like computational model in which tasks continue for the entire life of the distributed program. MPI built upon much of the experience in using PVM and is widely used for HPC platforms because most HPC platforms provide a high performance, native MPI implementation. Hence, for heterogeneous systems or for problems with dynamic process behavior, PVM is preferred. Otherwise, MPI is the most common communications library used for parallel processing. Efforts to extend MPI to support dynamic process creation and destruction have not been successful, although an early implementation of PVMPI was completed. PVM or MPI are used in our work for parallel applications.

The NetSolve project is being developed at the University of Tennessee's Innovative Computing Laboratory. NetSolve is a client-server system that enables users to solve complex scientific problems remotely. The system allows users to access both hardware and software computational resources distributed across a network. NetSolve searches for computational resources on a network, chooses the best one available, and using retry for fault-tolerance solves a problem, and returns the answers to the user. A load-balancing policy is used by the NetSolve system to ensure good performance by enabling the system to use the computational resources available as efficiently as possible

The problem description file (PDF) is the mechanism through which Netsolve enables services for the user. The pdf has been written which explains the problem .The pdf also explains the inputs and output files , the size of the matrix and the C code to which it links. Servers are introduced to the Netsolve agent and indicate the collection of programs that they support. Clients wishing to use Netsolve ask the agent to supply a list

of the best machines available for their particular problem. The client will then interact with the chosen server to perform the problem.

Netsolve is a good fit for HPRC applications at this point because the FPGA bitstreams for the RC elements are not portable to other RC boards of FPGA families. The most practical way to support HPRC applications with the grid is to use Netsolve. Research in to the performance of Netsolve, its effectiveness for HPRC, and how to extend Netsolve to provide any necessary extensions is discussed later in the report.

Reconfigurable Computing Development

Reconfigurable computing comes from a broader application of programmable logic device technology. Programmable logic devices provide designers with the ability to modify circuit functionality after its fabrication. Examples of programmable logic devices include programmable arrays of logic (PALs), complex programmable logic devices (CPLDs), field programmable system level integrated circuits (FPSLICs), and field programmable gate arrays (FPGAs), with FPGAs the most commonly used devices for reconfigurable computing.

Currently, reconfigurable computing systems are typically comprised of processing nodes associated with co-processing boards with FPGAs. This co-processing model uses software executing on the processor for general computation and control, with the reconfigurable hardware accelerating computational intensive operations. Following some promising research results, the emerging system on a chip market will likely provide SoCs with configurable logic.

Computing systems, particularly those employed in embedded applications, are increasingly benefiting from the use of programmable logic. Many developers use programmable logic devices (PLDs) to rapidly develop customized circuitry for various applications, and then ship the systems with the configured programmable logic devices. FPGAs used to be widely used for system prototyping or implementing glue logic on boards. The reduced device costs, increased capacities, faster operational speeds, and shrinking product time to market requirements now result in the common deployment of FPGAs in embedded systems as a substitute for ASICs. Such systems typically have a static configuration throughout their lifetime.

In some cases, the configurations of the programmable logic devices can be updated after the system is fielded, resulting in a configurable computing system. This ability to configure a system can reduce maintenance costs and extend product lifetimes via system upgrades in the field accomplished by updated configurations. In contrast, reconfigurable computing systems frequently change the programming of the logic devices during operation. Recent developments in programmable logic device technology (density, speed) have made reconfigurable computing practical.

Reconfigurable computers provide the user with the ability to dynamically change the logical operation of its computational elements. Configurable computers also provide the

ability to change the logical operation of computational elements, but in a more constrained manner.

Reconfigurable computing promises to provide a wide variety of benefits to system designers. A primary reason for the initial use of programmable logic devices is the relative ease in employing them in designs. Configuration requires seconds to perform, as opposed to the weeks required for fabricating an ASIC. If there are errors in the logic implementation, attractive system improvements, or changes to the specification, designers can make changes as necessary before the system is deployed. These benefits are equally applicable for configurable and reconfigurable systems.

The use of configurable or reconfigurable systems provides similar benefits in fixing errors, adding features, or providing other updates to systems throughout their life cycle. By modifying the functionality with a new configuration of the programmable logic, minimal update costs may be required. If one combines this model of providing upgrades or bug fixes with internet-connected configurable systems, radical improvements in the ability to maintain embedded systems become possible. Because reconfigurable computing provides this capability to modify system functionality to fix bugs or add features, cost reductions during design as well as after deployment result in significantly reduced life cycle costs. For products with long lifetimes, this cost reduction can be substantial.

The use of programmable logic results in accelerated development cycles yielding a reduced time to market. By avoiding the extended design and fabrication times required for ASICs, designers can reach the market more quickly. If design flaws or engineering change orders arise near the end of the design process, modifications to programmable logic can usually be implemented quickly with the existing parts. In the case of an ASIC-based design, a new version of the ASIC will typically be required, resulting in additional costs and delays. By avoiding these additional costs and delays, reconfigurable computing systems can reach the market more quickly. Because the timeliness in reaching the market often determines if a product is profitable or not, one cannot overstate the importance of the schedule risk mitigation due to reconfigurable hardware.

The flexibility coming from the ability to reconfigure components of a system based on environmental or operational conditions enables systems to be fielded for a broad range of applications and for dynamic or unknown environments. A single reconfigurable computing system can be applied to image processing, signal processing, cryptographic, and string matching problems through reconfigurations. An ASIC-based hardware approach would require a large investment in hardware resources while a software solution may not achieve the required performance. In this context, the flexibility of reconfigurable computing provides unmatched capabilities.

By optimizing the hardware to address the specific task at hand, reconfigurable computing platforms can often perform operations using much less power than general purpose hardware or software based solutions. Even custom or specialized hardware may not provide the power savings obtainable from customizing the hardware used to the

specific application. For example, in a number of signal and image processing applications, variable word sizes can be implemented with programmable logic to provide sufficient accuracy but no more. Reconfigurable computers can reduce the number of gates used in the design by optimizing word sizes as needed, whereas digital signal processors, microcontrollers, or general purpose processors must be designed with a static word size that can not be changed to reduce power.

Reconfigurable computers can support a very large number of gates through the use of multiple configurations. In effect, a given set of programmable logic devices can replace a much larger collection of ASICs. This will result in a reduction in the physical size required for the electronics comprising embedded systems.

Programmable logic devices provide an ideal capability to detect, identify, and isolate faulty hardware elements in embedded systems. With the ability to support testing configurations and to adapt to detected faults, reconfigurable computers can support a variety of critical applications. By enabling graceful system degradation in the presence of faults, much cheaper and more reliable critical systems can be fielded.

Last, and certainly not least, designers can achieve higher performance by using reconfigurable computing to adapt to the environmental conditions during operation. In one recent cryptography example, engineers achieved higher performance with an FPGA-based reconfigurable computing solution than available with an ASIC developed for the same task after a long and expensive development [33].

With reconfigurable computing systems, key attributes that determine its capabilities include the speed and granularity of reconfiguration. The ability to reconfigure programmable logic quickly makes it practical to change configurations more frequently during operation. The technology used in the programmable logic device dictates the granularity of reconfiguration that can most effectively be supported. The ability to selectively reconfigure parts of the programmable logic is known as fine-grain reconfiguration. Many families of programmable logic devices do not support the random access and modification necessary to support fine-grain reconfiguration. For these parts, most or all of the part must be reconfigured at the same time. Because the time to configure an entire part takes a large number of cycles, the performance penalty associated with such reconfigurations renders it impossible to support frequent function modifications while still attaining high performance. Hence, reconfigurable systems assembled from such parts have a coarse granularity of reconfiguration. In contrast, some newer generations of programmable logic devices support the fine-grain reconfigurability that enables hardware optimization based on the operations and data at hand.

Reconfigurable computers must include runtime systems to support the configuration data for the programmable logic device elements, much like a processor uses a sequence of instructions. For systems with slower, coarse-grain configuration, the programmable logic devices may be updated when each new application is run, or during changes between operational phases. As the frequency of reconfigurations increases and their

granularity decreases, the runtime support required becomes more sophisticated. Memory banks or configuration caches may hold the necessary configuration data, as with instruction and data caches. Although in principal, dynamic optimization of configurations is possible via the runtime system, in practice current limitations in the design tools currently make it impractical. Work with the Xilinx Jbits interface is improving the runtime and design environment to make such systems practical in the future. This research does not directly address runtime reconfiguration or configuration caching, but the performance modeling framework is intended to support these capabilities with little, if any, modification.

Reconfigurable computers bring together aspects of both hardware and software systems. Not surprisingly, debate rages about the best design languages, methodologies, and tools for reconfigurable computing systems. Many of the same issues and arguments concerning systems design and hardware/software codesign are applicable.

Most development efforts to map applications onto reconfigurable computers uses VHDL or Verilog for capturing the design, typically at the register transfer level. In doing so, hardware designers can use the same design capture, simulation, and synthesis languages and tools already used for ASIC development. In practice, the productivity from directly using HDLs lags behind industry needs. Designers write much of the HDL code at RTL, and too often do not employ language constructs such as VHDL generics, configurations, and generate statements to create portable, flexible designs. In addition, the synthesis tools provide roughly equivalent capability for FPGAs as with ASICs, enabling the reuse of much of ASIC design flows and tools.

The same domain specific attributes that make hardware description languages effective for designing electronic systems prove to be a significant limitation to the widespread adoption of VHDL or Verilog for capturing designs intended for reconfigurable computers. Software and systems engineers are not familiar with these hardware description languages and resist using them.

At the system design level, a number of proposed extensions to C or C++ have been forwarded by various companies to address behavioral design. Because C/C++ is widely used by systems engineers to develop system prototypes or executable specifications, it is hoped providing a facility to develop hardware designs in some C/C++ dialect will improve productivity and bring systems and hardware engineers closer together. Adoption of a C/C++ dialect potentially will potentially enable a much larger pool of designers to describe hardware because C/C++ users dwarf the HDL user population. The amount of infrastructure required with these C/C++ extensions may approach or even exceed that of using HDLs.

In an attempt to leverage the surging popularity of the Java programming language, as well as its support for code portability and for reuse via object-oriented facilities, researchers at BYU developed JHDL (see <http://www.jhdl.org>). The JHDL approach exploits the explosion in software development tools for Java and the much larger population of Java programmers to ease in the general adoption of reconfigurable

computing. JHDL lowers many of the barriers to entry for potential developers, and significantly simplifies the mapping of functionality between hardware and software. Nonetheless, performance limitations for Java hinder its adoption for high-performance applications.

A number of challenges remain for developers using reconfigurable computers. The verification of RC systems faces all the challenges of ASIC verification combined with the additional complexity of multiple configurations. Fine-grain adaptation of configurations provides a higher bar for systems verification.

Design languages, tools, and methodologies for RC systems continue to be a topic of research and development. The same productivity gap between available gates and designed gates currently hindering ASIC designers affects RC designers as well. Tools that support more abstract design while automatically extracting a problem's parallelism and the most appropriate configurations are needed. Addressing this problem is the focus of this research.

For software engineers, the notion of an instruction set architecture greatly reduces the difficulty in developing applications. This abstraction of a processor provides sufficient insight into the hardware to enable its effect use without overwhelming the programmer. Currently, reconfigurable computers lack such a standard "programmable configuration architecture" to serve a role like that of an instruction set architecture. This lack of a programmable configuration architecture could pose significant portability problems for the industry in the future.

Understanding the impact of architectural changes is just beginning; the changes to models of computation are not well understood yet. This remains as an exciting area of basic research with wide-reaching implications. For example, just as the notion of virtual memory has resulted in significant improvements in software, the notion of the virtualization of hardware will provide similar benefits for hardware design.

Mathematical Foundations

Performance analysis and architecture design for HPC and RC systems are challenging enough in their individual environments. For the proposed HPRC architecture, these issues and their interaction are potentially even more complex. Given these observations, it is evident that a mathematical modeling framework is a necessary tool for analyzing various performance metrics. Although substantial performance analysis research exists in the literature with regard to High Performance Computing (HPC) architectures [1,3,7,14,17,26,31,32,34,35,36,43] and even some with respect to Reconfigurable Computing (RC) [8,9,34], the analysis of these architectures working together has received little attention to date and currently there is a gap in the performance analysis research with regard to an HPRC type of architecture.

A modeling framework covering the common metrics found in HPC and RC research will fill this gap enabling performance analysis of the architecture and the analysis design

tradeoffs. The development of a modeling framework for a complex architecture such as HPRC presents several challenges and questions which will need to be addressed: *modeling communication time* (node-to-node, processor-to-RC unit, and RC unit-to-RC unit), *modeling computation time* (software in processor and firmware in RC unit), *modeling setup costs* (application setup, RC unit configuration, and network configuration), and *modeling load imbalance* (application tasks or data imbalance and hardware versus software imbalance).

HPC performance is commonly measured in terms of speedup and efficiency. The basic definition for *speedup* in HPC is the ratio of the execution time of the best possible serial algorithm on a single processor to the parallel execution time of the parallel algorithm on an m -processor parallel system:

$$S(m) = \frac{R(1)}{R(m)} \quad \text{where } R \text{ represents the execution time} \quad (1)$$

Efficiency is defined as the ratio of speedup to the number of processors, m :

$$Eff(m) = \frac{S(m)}{m} = \frac{R(1)}{m \cdot R(m)} \quad (2)$$

We can also discuss *speedup* and *efficiency* for reconfigurable computing. Rather than processing nodes running software algorithms in parallel on a common problem as in HPC, in RC we have hardware and software algorithms working together on a common problem. In general, we will define *RC speedup* as the ratio of the execution time of the best possible software only algorithm on a single processor to the execution time of the RC algorithm executing in software and hardware:

$$S_{RC} = \frac{R_{software}}{R_{RC}} \quad \text{where } R \text{ represents the execution time} \quad (3)$$

We should note that depending on the algorithm and implementation, there may be a hardware/software load imbalance. We will discuss this further in a later section but this essentially means that the execution may have a percentage of hardware/software overlap ranging from fully parallel (both are busy 100% of the time) to no overlap where the passing from software to hardware is essentially serial. A higher percentage of overlap will result in the greatest potential for speedup leading to the idea of efficiency. The definition of *RC efficiency* will be slightly different from HPC efficiency. For HPC, we have software tasks divided among a number of processing nodes all working in parallel to produce some amount of speedup. The efficiency is a result of this speedup relative to the overhead induced from the number of nodes working in parallel. For RC, we have k software tasks, and one or more hardware tasks, n . The efficiency will be the speedup achieved relative to the additional RC system overhead (setup, communication, configuration, etc.). We will assume this overhead is related to the total number of hardware and software tasks even though these tasks are not necessarily performing the same functions, resulting in the following definition for *RC efficiency*:

$$Eff_{RC} = \frac{S_{RC}}{n+k} = \frac{R_{software}}{(n+k) \cdot R_{RC}} \quad (4)$$

Again for HPRC performance metrics, we will consider *speedup* and *efficiency*. Our general definition for *HPRC speedup* is the ratio of the execution time of the best possible serial software only algorithm executing on a single processor to the parallel execution time of the parallel HPRC algorithm executing in software and hardware at some load balance on an m -node system (where a node is a workstation with or without RC units):

$$S_{HPRC}(m) = \frac{R(1)}{R_{HPRC}(m)} \quad \text{where } R \text{ represents the execution time} \quad (5)$$

We will discuss the details of the load balance implications in a later section. Similar to the case for RC efficiency, we define *HPRC efficiency* where n is the total number of hardware tasks and k is the total number of software tasks across all nodes of the system:

$$Eff_{HPRC}(m) = \frac{S_{HPRC}(m)}{n + k} = \frac{R(1)}{(n + k) \cdot R_{HPRC}(m)} \quad (6)$$

Other metrics such as power, cost, physical size, etc. are important especially in embedded systems. For these metrics, we can form cost functions from the performance analysis equations and optimize the metric as appropriate.

Overview

To effectively use the proposed HPRC architecture, we must be able to analyze design tradeoffs and evaluate the performance of applications as they are mapped onto the architecture. Performance models are commonly used tools for analyzing and exploiting available computational resources in HPC environments. Some commonly used modeling techniques in the analysis of computing systems are analytic models, simulations, and measurements. The best suitable modeling approach depends on the required accuracy, level of complexity, and analysis goal of the model. We can employ one or more of these modeling approaches to better understand the tradeoffs in mapping applications to HPRC resources as well as the most effective ways of doing so.

To develop a representative modeling framework for HPRC we will begin by investigating and characterizing the RC architecture and expanding this model to multiple nodes representative of an HPRC platform. In the RC environment, the focus will be on FPGA configuration, processor to FPGA communication, data distribution between FPGA and processor, memory access time, computation time in hardware and software, and other RC application setup costs. Next, we apply this knowledge to the multi node environment building on the earlier load balance work by Peterson [35]. We will develop an analytic modeling methodology for determining the execution time of a synchronous iterative algorithm and the potential speedup. *Synchronous iterative algorithms*, present in a large class of parallel applications, are iterative in nature and each iteration is separated from the previous and subsequent iterations by a synchronization operation. Examples of synchronous iterative algorithms include simulations and many image processing and data classification algorithms.

What follows below is the first iteration of the model using our available hardware and firmware. Table 1 lists the symbols and definitions used in this section.

Table 1. Symbols and Definitions

Symbol	Definition		Symbol	Definition
m	Number of workstations		n	Number of hardware tasks
S_P	Speedup		t_{RC}	Time for a parallel hardware/software task to complete
R_P	Runtime on parallel system		t_{RC_work}	Total work performed in hardware and software
R_{RC}	Runtime on RC system		t_{avg_task}	Average completion time of hardware or software task on RC system in a given iteration
I	Number of iterations		β	Load imbalance factor between hardware and software
$t_{overhead}$	Iteration overhead operations		σ	Hardware acceleration factor for RC system
t_{SW}	Time to complete software tasks		R_I	Runtime on a single processor
t_{HW}	Time to complete hardware tasks		r	Number of hardware tasks not requiring new configuration
$t_{parallel, RC}$	Time to complete parallel task on multi node system		d	Number of hardware tasks not requiring new data set
t_P	Time to complete parallel host software tasks		t_{config}	Time for FPGA configuration
t_{work}	Total work performed in hardware and software on all nodes of a multi-node system		t_{data}	Time for data access
β_k	RC node Load imbalance factor at node k in a multi-node system		t_{host_serial}	Host serial operations
α	Load Imbalance factor between host nodes in a multi-node system		t_{node_serial}	RC node serial operations
σ_k	Hardware acceleration factor for node k in muti-node system		$t_{host_overhead}$	Iteration overhead operations for hosts
			$t_{node_overhead}$	Iteration overhead operations for RC nodes

Reconfigurable Computing Node Analysis

Our performance model analysis will begin with a single RC node running a synchronous iterative algorithm. These restrictions will allow us to investigate the interaction between the processor and RC unit.

First, we will assume we have a segment of an application that has I iterations and all iterations are roughly the same as shown in Figure 2. The RC unit has at least one FPGA (there may be other reconfigurable devices which provide control functions) and tasks can potentially execute in parallel in hardware (RC unit(s)) and in software on the processor. We should point out that the hardware and software task trees can be arbitrarily complex. Figure 2 shows a simple hardware/software tree structure. Additionally, hardware can be reused within a given iteration if the number of tasks or size of the task exceeds the number of available FPGAs.

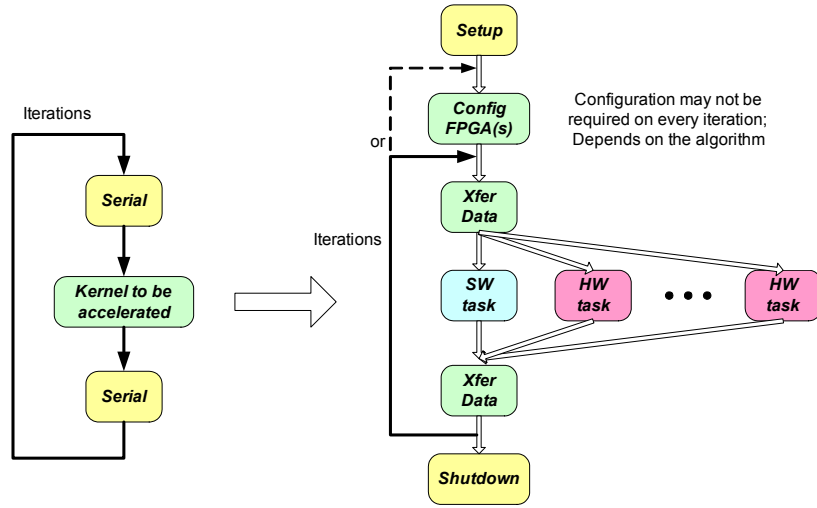


Figure 2. Synchronous Iterative Algorithm

For a synchronous iterative algorithm, the time to complete a given iteration is equal to the time for the last task to complete either in hardware or software. For each iteration of the algorithm, there are some operations which are not part of the kernel to be accelerated and are denoted $t_{serial,i}$. Other overhead processes that must occur such as configurations and exchange of data are denoted $t_{overhead,i}$. The time to complete the kernel tasks executing in software and hardware are $t_{SW,i}$ and $t_{HW,i}$, respectively. For I iterations of the algorithm where n is the number of hardware tasks, the runtime, R_{RC} , can be represented as [35]:

$$R_{RC} = \sum_{i=1}^I [t_{serial,i} + \max(t_{SW,i}, \max_{1 \leq j \leq n} [t_{HW,i,j}]) + t_{overhead,i}] \quad (7)$$

To make the math analysis cleaner, we will make a couple of reasonable assumptions. First, we will assume that each iteration requires roughly the same amount of computation allowing us to remove the reference to individual iterations in equation (7).

Second, we will model each term as a random variable and use their expected values. Thus we define t_{serial} as the expected value of $t_{serial,i}$ and $t_{overhead}$ as the expected value of $t_{overhead,i}$. The mean time required for the completion of the parallel hardware/software tasks is represented by the expected value of the maximum (t_{SW}, t_{HW}). Finally, we will assume that each of the random variables are independent and identically distributed (iid). We can then write the run time as:

$$\begin{aligned} R_{RC} &= \sum_{i=1}^I (E[t_{serial,i}] + E[\max(t_{SW,i}, \max_{1 \leq j \leq n} t_{HW,i,j})] + E[t_{overhead,i}]) \\ &= I(t_{serial} + E[\max(t_{SW}, \max_{1 \leq j \leq n} t_{HW,j})] + t_{overhead}) \end{aligned} \quad (8)$$

If the iterations are not iid, we must retain the first form of equation (8) and the math analysis is more difficult.

We can rewrite the RC system hardware/software tasks in terms of the total work. We will assume that all hardware tasks are the same and that the time to complete a software tasks is the same as that for a hardware task. With these simplifying assumptions, we will define $E[t_{avg_task}]$ is the expected average task completion time (including hardware and software tasks) within an iteration for the RC system. Therefore the total work completed on the RC system, measured in runtime is given by:

$$t_{RC_work} = E[t_{SW} + \sum_n t_{HW}] = (n+1) \cdot E[t_{avg_task}] \quad (9)$$

The division of tasks between hardware and software creates an *RC load imbalance* [35] which we will represent as β . Considering the affect of the imbalance on the task completion time, the expected value of the maximum task completion time can be expressed as the average task time within an iteration multiplied by the RC load imbalance factor:

$$E[\max(t_{SW}, \max_{1 \leq j \leq n} t_{HW,j})] = \beta \cdot E[t_{RC_system}] \quad (10)$$

Combining equations (9) and (10) we can rewrite the maximum task completion time as,

$$E[\max(t_{SW}, \max_{1 \leq j \leq n} t_{HW,j})] = \frac{\beta \cdot t_{RC_work}}{n+1} \quad (11)$$

Note that if the load is perfectly balanced β is the ideal value of 1 and the total work is divided equally among the hardware and software tasks. If the tasks are performed serially (no concurrent hardware/software operation), β is the worst case value of $(n+1)$, where n is the number of hardware tasks. Thus as the load imbalance becomes worse, β increases ranging from a value of 1 to $(n+1)$.

Noting that the total work measured in time for a software-only solution is not equivalent to the total work measured in time on an RC system solution, we introduce an acceleration factor σ to account for the difference. Since the goal of RC systems is to speed up an application, only tasks that would be faster in hardware are implemented in hardware. For example, an FFT in software may take longer to execute than an equivalent implementation in the hardware. Given the total work that will be completed

in hardware and software on an RC system, we can represent the software only run time on a single processor as:

$$R_1 = I \cdot (t_{serial} + t_{SW} + \sigma \cdot \sum_n t_{HW}) \quad (12)$$

The overhead for an RC system consists of the FPGA configuration time and data transfer time. The configuration time for the FPGA(s) is $(n-r) \times t_{config}$, where r is the number of hardware tasks not requiring a new configuration. The time to transfer data to and from the RC unit is $(n-d) \times t_{data}$, where d is the number of hardware tasks not requiring a new data set.

The speedup, S_{RC} , is defined as the ratio of the run time on a single processor to the run time on the RC node:

$$S_{RC} = \frac{R_1}{R_{RC}} = \frac{t_{serial} + t_{SW} + \sigma \cdot \sum_n t_{HW}}{t_{serial} + \frac{\beta \cdot t_{RC_work}}{n+1} + [(n-d) \cdot t_{data}] + [(n-r) \cdot t_{config}]} \quad (13)$$

Using equation (13) we can investigate the impact of load imbalance and various overhead issues on the algorithm performance by varying β , t_{config} , t_{data} , t_{RC_work} , r , d , and n .

High Performance Reconfigurable Computing Multi-Node Analysis

Now that we have a model for a single RC node and an understanding of the basic HPC issues involved in a set of distributed nodes, we will turn our focus to expanding the model for multi-node analysis. An example of the HPRC architecture was shown in Figure 1. For now, we will not consider the optional configurable interconnection network between the RC units in our modeling analysis.

We will again start our performance model analysis using a synchronous iterative algorithm this time running on a platform consisting of multiple RC nodes. The restriction of synchronous iterative algorithms will allow us to investigate the communication and synchronization that occurs among nodes between iterations. We will begin our model by restricting our network to a dedicated homogeneous system where there is no background load (i.e. all nodes are identical, same processor and same RC system configuration).

Again, we will assume we have a segment of an application having I iterations that will execute on parallel nodes with hardware acceleration. Additionally, we will assume that all iterations are roughly the same as is shown in Figure 3. Software tasks can be distributed across computing nodes in parallel and hardware tasks are distributed to the RC unit(s) at each individual node.

For a synchronous iterative algorithm, the time to complete an iteration is equal to the time for the last task to complete on the slowest node whether it be hardware or software.

For each iteration of the algorithm, there are some calculations which cannot be executed in parallel or accelerated in hardware and are denoted $t_{master_serial,i}$. There are other serial operations required by the RC hardware and they are denoted $t_{node_serial,i}$. Other overhead processes that must occur such as synchronization and exchange of data are denoted $t_{master_overhead,i}$ and $t_{node_overhead,i}$ for the host and RC systems respectively. The time to complete the tasks executing in parallel on the processor and RC unit are $t_{SW,i,k}$ and $t_{HW,i,j,k}$ respectively. For I iterations of the algorithm where n is the number of hardware tasks at node k and m is the number of processing nodes, the runtime, R_P , can be represented as [35]:

$$R_P = \sum_{i=1}^I [t_{master_serial,i} + t_{node_serial,i} + \max_{1 \leq k \leq m}(t_{SW,i,k}, \max_{1 \leq j \leq n}[t_{HW,i,j,k}]) + t_{master_overhead,i} + t_{node_overhead,i}]$$

$$R_P = \sum_{i=1}^I [t_{master_serial,i} + t_{node_serial,i} + \max_{1 \leq k \leq m}(t_{RC,i,k}) + t_{master_overhead,i} + t_{node_overhead,i}] \quad (14)$$

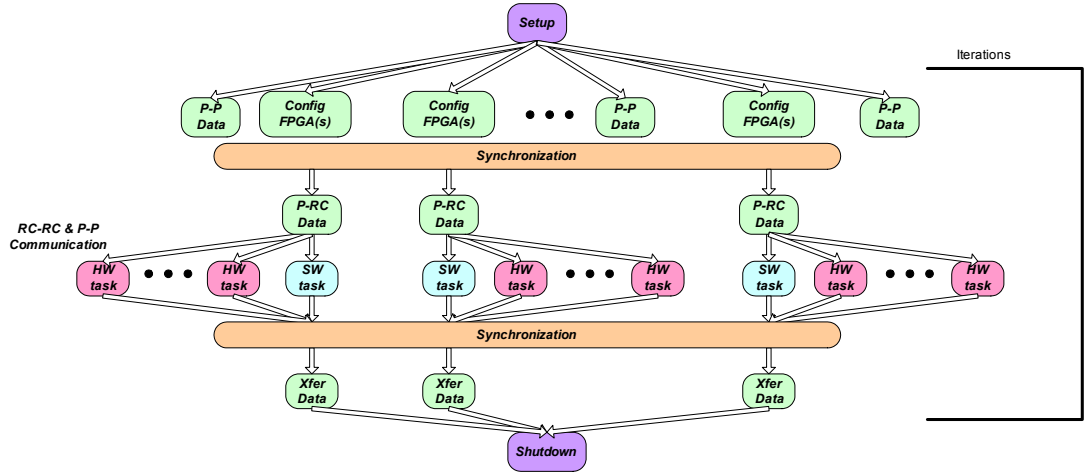


Figure 3. Flow of Synchronous Iterative Algorithm for Multi Node Analysis

Again, to make the math analysis cleaner, we will make a couple of reasonable assumptions. First, we will assume that each iteration requires roughly the same amount of computation thus we can remove the reference to individual iterations in equation (14). Second, we will also assume that each node has the same hardware tasks and configuration making the configuration overhead for each node the same. Third, we will model each term as a random variable and use their expected values. Thus we define t_{master_serial} and t_{node_serial} as the expected value of $t_{master_serial,i}$ and $t_{node_serial,i}$. Similarly, we define $t_{master_overhead}$ and $t_{node_overhead}$ as the expected value of $t_{master_overhead,i}$ and $t_{node_overhead,i}$. The mean time required for the completion of the RC hardware/software tasks is represented by the expected value of the maximum $t_{RC,k}$ ($1 \leq k \leq m$). Finally, assuming that the random variables are each independent and identically distributed (iid), the run time can then be expressed as:

$$R_P = \sum_{i=1}^I (E[t_{master_serial,i}] + E[t_{node_serial,i}] + E[\max_{1 \leq k \leq m}(t_{RC,i,k})] + E[t_{master_overhead,i}] + E[t_{node_overhead,i}]) \quad (15)$$

$$R_P = I(t_{master_serial} + t_{node_serial} + E[\max_{1 \leq k \leq m}(t_{RC,k})] + t_{master_overhead} + t_{node_overhead})$$

As discussed in section 0, we can rewrite the total work at node k in terms of the average task completion time rather than the maximum (see equation (9)). Again assuming the random variables are iid, from equation (9) we can express the total work across all m nodes in the HPRC platform as:

$$t_{work} = \sum_{k=1}^m t_{RC_work,k} = \sum_{k=1}^m (n+1) \cdot E[t_{RC_system,k}] = m \cdot (n+1) \cdot E[t_{RC_system}] \quad (16)$$

As discussed earlier for RC systems, when tasks are divided a load imbalance exists. We will represent the RC load imbalance at a particular node k as β_k . Additionally, there exists an application load imbalance across all of the nodes of the HPRC platform. We will represent this node-to-node load imbalance as α . We will assume the host node application load imbalance α and RC system load imbalance β_k are independent. Additionally, we will assume that the RC system load imbalance at any node is independent of the others. The completion time can then be expressed as the average task completion time within an iteration multiplied by the load imbalance factors (which together have a multiplicative effect):

$$E[\max_{1 \leq k \leq m}(t_{RC,k})] = \alpha \cdot \beta_k \cdot E[t_{RC_system}] \quad (17)$$

Combining equations (16) and (17) we can rewrite the maximum task completion time as,

$$E[\max_{1 \leq k \leq m}(t_{RC,k})] = \frac{\alpha \cdot \beta_k \cdot t_{work}}{m \cdot (n+1)} \quad (18)$$

Note that if the RC load is perfectly balanced or if the algorithm runs entirely in software ($n=0$), β_k is the ideal value of 1. As the RC load imbalance becomes worse, β_k increases. Similarly, if the node-to-node load is perfectly balanced or if the algorithm runs entirely on a single node, $m=1$, α is the ideal value of 1 and the model reduces to the model for a single RC node as in equation (11). Finally, as the node to node imbalance becomes worse, α increases.

Noting that the total work measured in time for a software-only solution is not equivalent to the total work measured in time on an HPRC platform solution, we introduce an acceleration factor σ_k to account for the difference at each node k . Given the total work that will be completed in hardware and software on an HPRC platform, we can represent the software only run time on a single processor as:

$$R_1 = I \cdot [t_{master_serial} + \sum_{k=1}^m (t_{SW} + \sigma_k \cdot \sum_n t_{HW})] \quad (19)$$

The overhead for the HPRC platform consists of the FPGA configuration and data transfers as discussed in section 0 and the synchronization between the nodes. We will model the time required for synchronization as a logarithmic growth with the number of

nodes [35]. The speedup, S_p , for the HPRC platform is defined as the ratio of the run time on a single processor to the run time on m RC nodes:

$$S_p = \frac{R_1}{R_p}$$

$$= \frac{t_{master_serial} + \sum_{k=1}^m (t_{SW} + \sigma_k \cdot \sum_n t_{HW})}{t_{master_serial} + t_{node_serial} + \frac{\alpha \cdot \beta_k \cdot t_{work}}{m \cdot (n+1)} + [(n-d) \cdot t_{data}] + [t_{synch} \cdot \log_2[m]] + [(n-r) \cdot t_{config}]} \quad (20)$$

Using equation (20) we can investigate the impact of load imbalance and various other overhead issues on the algorithm performance by varying σ_k , α , β_k , t_{synch} , t_{config} , t_{data} , m , and n .

High Performance Reconfigurable Computing Development Hardware

There are two HPRC clusters that are available for developing and validation of the model. The Air Force Research Laboratory in Rome, NY is assembling a “heterogeneous HPC” which is a cluster of four Pentium nodes populated with Firebird boards. Future plans include expanding to more nodes. The HPRC cluster at UT consists of eight Pentium III nodes populated with Pilchard boards. The Pilchard boards include a Xilinx Virtex 1000E FPGA with approximately one million gate capacity. Each Pilchard board is placed in a DIMM slot to exploit the memory bus speeds, thus providing significantly faster communications than that achieved by PCI-based RC boards. Other available hardware for parameter measurements includes a cluster of Sun workstations, Wildforce, Firebird and SLAAC RC boards. The Air Force Research Lab is also acquiring a large HPRC platform with approximately 50 nodes, each with FPGAs boasting a total capacity of over 10 million gates. The nodes are to be interconnected via Myrinet, thus the Air Force will have a large HPRC platform with half a billion gate capacity.

Reconfigurable Computing Model

Developing a cost metric should be straightforward based on the processors, FPGAs, memory, and interconnect under consideration. A cost function can be developed relating execution time or speedup as determined from the proposed model to the cost of the system. Similarly a power metric cost function could be developed relating execution time determined from the proposed model to the total power used by the system.

We have made basic model parameter measurements using a sample application for the Wildforce board as a benchmark. To validate the execution time prediction of the model, the benchmark measurements are used with the developed model to predict the runtime for three of the CHAMPION demos. The details of the demo applications will be discussed in a later section. From the benchmark, we have determined the model parameters as shown in Table 2. The configuration values for CPE0 and PE1 are significantly different because they are two different Xilinx devices and we therefore

account for them separately in the model calculations. For this application the only part of the algorithm considered as serial is the board configuration and setup. There is only one iteration therefore I is one. The remaining unknowns are the values for the total work and the application load imbalance.

Table 2. Model Parameters for Wildforce from Benchmark Application

	CPE0	PE1	HW	Data	Setup (tsw)	Serial
Average in usec	535275	257232.8	1250.52	33282.08	68892.34	40750.46

The total work can be determined from the amount of work completed by the software task plus the amount completed by the hardware task. This can be represented in terms of the number of events multiplied by the execution time per event:

$$t_{RC_work} = N_e \cdot t_{hw_exe} + t_{sw_exe} \quad (21)$$

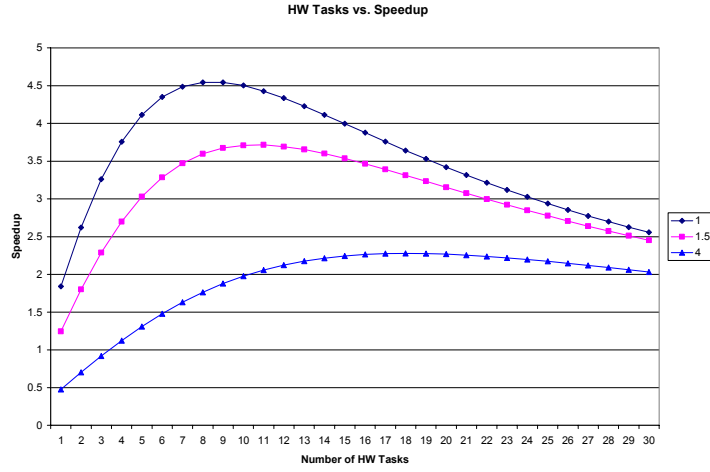
Where N_e is the total number of events and t_{sw_exe} is t_{hw_exe} are the software and hardware execution times per event respectively. In this particular application, the software and hardware tasks do not overlap so the application load imbalance, β , would be the maximum worst-case value of $(n+1)$ or 2 in this case.

Using the denominator of equation (13), we can use the model to predict the runtime of the CHAMPION demo algorithms. The average runtime of fifty trials on the Wildforce RC system is shown in Table 3 and Figure 5 along with the model predictions. The number following the algorithm name indicates the input data size. The value 128 indicates an input data set of 128x128 and similarly the value 256 indicates a 256x256 input data set.

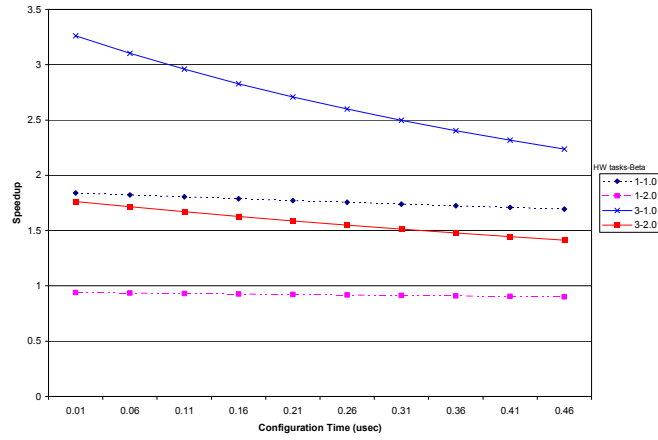
One possible candidate for the error in the model prediction is our measurement techniques for the model parameters. We believe that the measurements have enough accuracy and the only problem could be in the overhead introduced by the probes contributing to the over estimation of the runtime.

Another possible error contributor is the model methodology and assumptions. Being the first pass at modeling the performance of an RC system, the representation for the total work and application workload balance may be inaccurate. More studies of different algorithms and systems will be required to make a final determination on the accuracy of this representation.

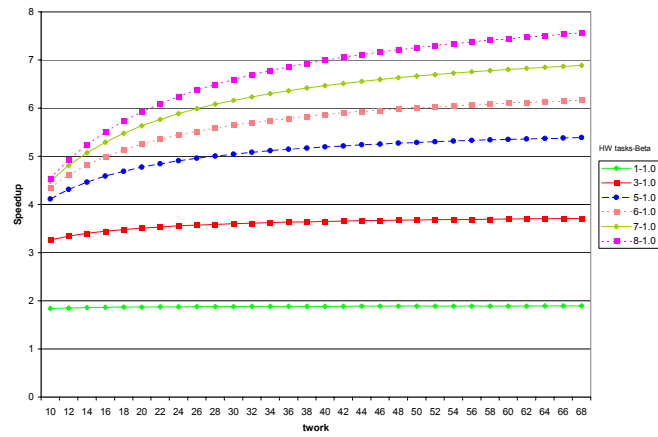
Finally, issues not represented in the model may be contributing to the error. System issues such as caching, optimum data packet size, and other optimization techniques used in operating systems and possibly the RC board API will need to be investigated.



(a) Vary number of RC units



(b) Vary RC configuration time



(c) Vary amount of total work

Figure 4. Speedup Curves: a) Vary number of RC units, b) Vary configuration time, and c) Vary total work

Table 3. Runtime Predictions and Measurements (time in seconds)

	model prediction	average
hipass_128	0.911342	1.313353
hipass_256	1.773769	1.907098
START_128	5.166674	4.597426
START_256	6.175542	6.121883
START20_128	6.702268	8.134971
START20_256	7.741632	8.855299

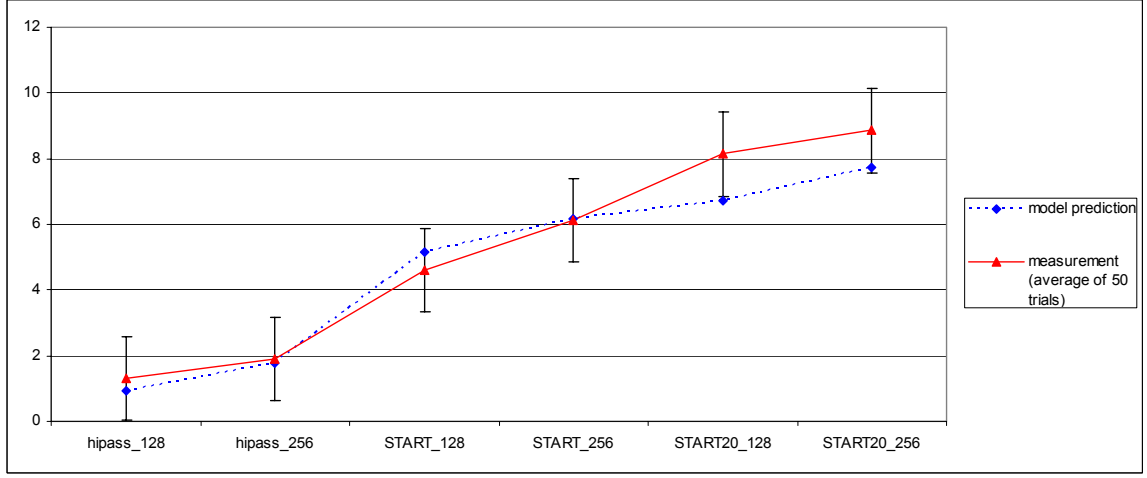
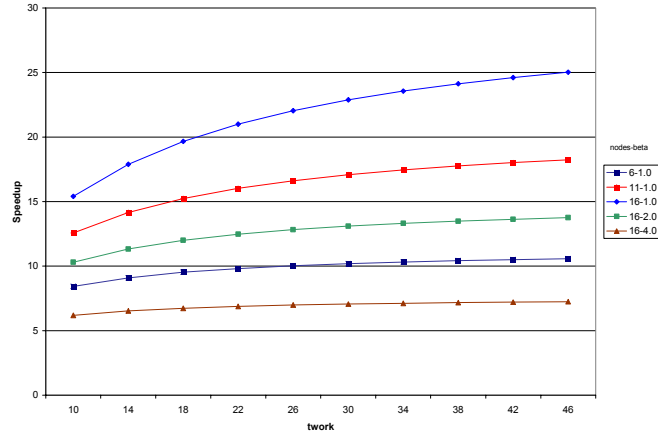


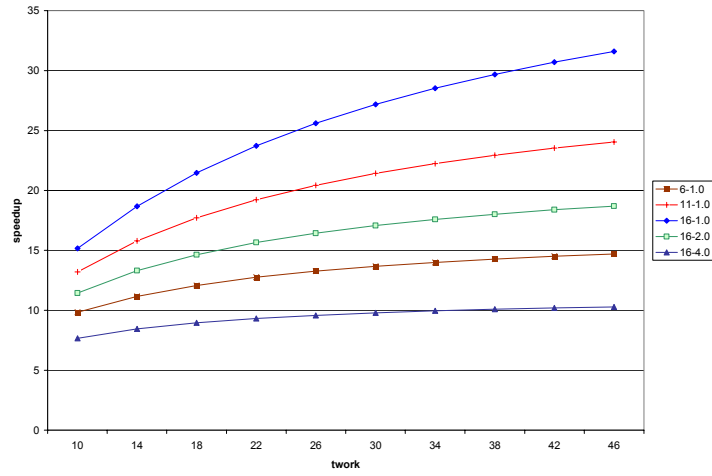
Figure 5. Comparison of RC Model Prediction with Measurement Results

High Performance Reconfigurable Computing Model

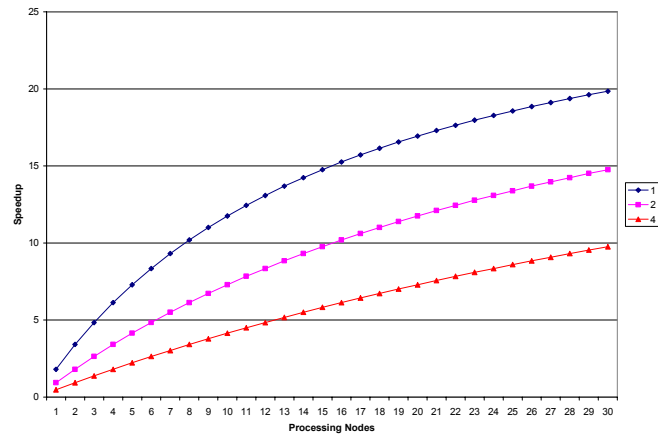
Again we will use the measured values from the Wildforce and Firebird experiments for the RC parameters. Figure 6 (a) and (b) show the speedup curves for various β_k values and number of nodes for an increasing workload (x-axis). The serial and configuration overhead are held constant at 0.2usec and 0.01usec per FPGA respectively. In figure (a) each node has one RC unit per node, and we plot speedup for increasing work for various load imbalance and node configurations. In figure (b) each node has two RC units per node, and again we plot speedup for increasing work for various load imbalance and node configurations. As seen in both figures, the speedup improves with increasing workload for lower β_k values. As β_k increases however, the speedup obtained is impacted by the load imbalance not only between nodes but also between processors and RC hardware. Thus, for a higher β_k value, higher node systems and/or those with more FPGAs per unit are more severely impacted by the load imbalance. In figure (c) we show the speedup for three load imbalance values with the amount of total work fixed as the number of nodes in the system increases. Each node has one hardware task.



(a) One RC unit per node (6, 11, and 16 nodes)



(b) Two RC units per node (6, 11, and 16 nodes)



(c) Fixed work, One RC unit per node

Figure 6. Speedup Curves: a) Vary work for one FPGA per RC unit and b) Vary work for two FPGAs per RC unit and c) Increasing number of nodes (one RC unit per node) work fixed

Model Applications

An obvious use of the proposed modeling framework is the performance evaluation of potential application mappings. With the proposed model, a variety of architectural configurations can be studied and compared without actually implementing them on the real system. The proposed model for the HPRC platform can also serve as the foundation for other task scheduling and load balancing cad tools. Another performance related use of the model, specifically the RC portion of the model, is as a method to classify the computing capability of an RC node which could be useful in the NetSolve [30] and SinRG [41] programs at the University of Tennessee.

Other interesting analysis problems possible with the proposed modeling framework are tradeoff studies of power, size, cost, network bandwidth, etc. For a given application, cost functions can be minimized for a fixed power budget or physical size for instance. The significance of this capability may be lost in traditional HPC environments, but in embedded systems where designs are often constrained by size and battery power, these issues are extremely important.

Given this model for execution time in an RC system we can begin to discuss the determination of a cost function for power. To determine a rough estimate of the system power, we must determine the power used by the processor and by the RC unit(s). Assuming our processor does not operate in sleep mode (i.e. the power consumption is constant with time) we can determine the total power consumption by the processor based on the execution time from the model and power information from the processor's literature. Determining the total power consumption for the RC unit(s) is somewhat more complicated. Power consumption in FPGAs is dependent on the speed and size of the design. The FPGA literature should provide some guidelines for determining power consumption based on the number of gates or logic blocks used in the design and the speed of the design. Once these factors are know, the total power for the RC unit(s) can be calculated using the execution time from the model. Finally, the total power consumption for the RC system is simply the sum of the total power consumed by the processor and the RC unit(s).

From the model equations and power parameters for the processors and reconfigurable devices, developing a cost function relating execution time to power consumed should be straightforward based on the configuration of the HPRC platform under consideration. The function can then be optimized for any of the architecture choices such as number of nodes or size of FPGA for example. A similar cost function could be developed relating execution time from the proposed model to the total cost of the system.

The modeling framework for the HPRC architecture is also applicable to the growing field of Systems on a Chip (SoC). Many parallels can be drawn between the issues encountered in SoC design and the architectural questions for HPRC. SoC is a self-contained electronic system residing on a single piece of silicon as shown in Figure 7. It is likely to contain a processor or controller, multiple-memory modules, analog or mixed signal circuitry, a system bus structure, hard and soft IP cores, and reconfigurable logic. With all of these different types of design styles to contend with, it is imperative that the

chip designer have a deep understanding, not only of the functionality of the different subsystems, but also of the complex interactions between the subsystems. SoC design requires front-end planning and feedback on system performance parameters at all stages of the design cycle. A modeling framework such as proposed for the HPRC architecture could provide this performance feedback to the designer. SoC processors and memory modules are very similar to the nodes in an HPRC system and both architectures can also incorporate reconfigurable units. Also, both architectures have a communication backbone in the form of a system bus or some sort of network. They are not only architecturally similar but also share some of the same design methodologies such as hardware/software codesign. With all these similarities, successful development of a framework for the HPRC architecture would also be an advantage for the SoC community.

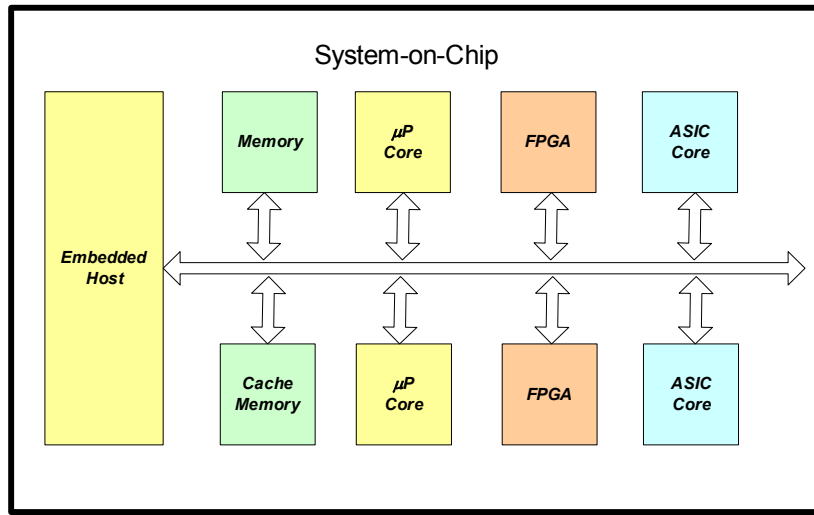


Figure 7. SoC Architecture Example

Initial analysis and parameter measurements have been conducted and the first editions of the RC and HPRC models have been developed. The parameter measurements for the RC platform need to be expanded to include tests for determining the load balancing parameters. Future experiments include parameter and validation measurements for the multi node environment. Models to predict the RC and application load imbalance factors are currently under development using the load balancing results from Peterson's work as a guide [35]. Also, cost functions for power and total cost of the system need to be fully modeled and verified. As the HPRC platform applications are developed and studied, the use of the model for scheduling and load balancing will be demonstrated as a manual exercise.

Efficiency is a common metric for HPC performance but is not easily transformed into the RC domain. We gave a preliminary analysis of the concept of efficiency in RC systems but more investigation and thought will be given to this concept as well as applying it to the HPRC platform. One idea currently being explored is the concept of an *effective processor* quantity that can be applied to the FPGA. If we can determine the effective processing capability or P_{eff} for a given FPGA in an RC system, the efficiency

concept from the HPC domain will more directly apply. A review of the work by Dehon [8] may be helpful in developing this quantity.

HPRC Application Examples

In order to evaluate the effectiveness of HPRC for accelerating applications, validate the performance modeling work for HPRC, and to investigate programming tools, languages, and computational models, a variety of application examples for HPRC have been developed.

Hyperspectral methods for deriving information about the Earth's resources using airborne or space-based sensors yield information about the electromagnetic fields that are reflected or emitted from the Earth's surface, and in particular, from the spatial, spectral, and temporal variations of those electromagnetic fields. Chemistry-based responses which are the primary basis for discrimination of the land cover types in the visible and near infrared portions of the spectrum are determined from the data acquired simultaneously in multiple windows of the electromagnetic spectrum. In contrast to airborne and space-based multispectral sensors, which acquire data in a few (<10) broad channels, hyperspectral sensors can now acquire data in hundreds of windows, each less than 10 nanometers in width. Because many land cover types have only subtle differences in their characteristic responses, this potentially provides greatly improved characterization of the unique spectral characteristics of each, and therefore increases the classification accuracy required for the detailed mapping of species from remotely sensed data.

Many image classification problems are characterized by a large number of inputs and moderately large number of classes that can be assigned to any input. Two popular simplifications have been considered for such problems: (i) *feature extraction*, where the input space is projected into a smaller feature space, thereby reducing the dimensionality and associated computation required, and (ii) *modular learning*, where a number of classifiers, each focusing on a specific aspect of the problem are learned instead of a single classifier. Several methods for feature extraction and modular learning have been proposed in the computational intelligence community.

Prediction of land cover types from airborne/space borne sensors is an important classification problem in remote sensing. Due to advances in sensor technology, it is now possible to acquire hyper spectral data simultaneously in more than 100 bands, each of which measures the integrated response of a target over a narrow window of the electromagnetic spectrum. The bands are ordered by their wavelengths and spectrally adjacent bands are generally statistically correlated with target dependent groups of bands.

Using such high dimensional data for classification of land cover potentially improves discrimination between classes but dramatically increases problems with parameter estimation and storage and management of the extremely large datasets.

To explore the potential for using HPRC to accelerate hyperspectral imaging tasks, we investigate a clustering algorithm called “k-means clustering” and its efficient implementation using HPC scalable architectures. This algorithm has its origin in the data-mining field. It is utilized for classification purposes and to discover anomalies and patterns in both small and large data sets. There exist many different variants of k-means clustering, most of which are variants adapted for special purpose environments [10,11,19,20,42,47]. With the growth of data collected on operational and transactional data, the field of data mining has become increasingly important. The growth of data has been accelerated with the commercialization of the Internet and the increased use of personal computer. In this environment collection of individual metrics is relatively cheap and unobtrusive to the user. Companies who have been collecting vast amount of data on consumer habits are now confronted with the dilemma of what to do with all the data. This is where k-means clustering becomes useful. It provides a remedy tailored to this problem and reveals patterns that otherwise are obfuscated and not otherwise discernible. In short, it can be said that k-means clustering is a common solution to the segmentation of multi-dimensional data. However, these large amounts of data sets require large computational capacity. The nature of this problem is ideally implemented on a High performance computing architectural node.

We now consider the k-means clustering algorithm. Given a set of N pixels, each composed of S spectral channels, and represented as a point in S -dimensional Euclidean space we partition the pixels into K clusters with the property that pixels in the same cluster are spectrally similar. The k-means clustering algorithms (there are several variants) provide an iterative scheme that operates over a fixed number (K) of clusters, while attempting to simultaneously optimize center locations and pixels assignments.

From an initial sampling, the algorithm loops over all the data points, and reassigns each to the cluster whose center is closest. After the pass through the data, the cluster centers are recomputed. Each iteration reduces the total within-class variance for the clustering, so it is guaranteed that after enough iterations, the algorithm will converge, and further passes will not reassign points.

For experimental and initial validation purposes, ten data points (pixels) with 2 dimensions have been considered for classification into 3 classes. The results obtained so far are preliminary, but the framework for continued research has been set up. These results are obtained from small algorithmic transformation. Points are assigned to the cluster centers to which they are the closest; for the minimum-variance criterion, “closest” is defined in terms of the distance. Consider a point x and cluster center c where “ i ” indexes the spectral components of each. The Euclidean distance is defined:

$$\|x - c\|^2 = \sum |x_i - c_i|^2$$

Other distance measures can also be used; for instance, the general family of p -metrics (for which the Euclidean distance is a special case $p = 2$) is given by

$$\|x - c\|^p = \sum |x_i - c_i|^p$$

In our case we use $p = 1$ to find the Manhattan distance. The Euclidean distance has several advantages. For one, the distance is rotationally invariant. Furthermore, minimizing the Euclidean distance minimizes the within-class variance. On the other

hand, the Euclidean distance is more expensive to implement in hardware since it involves multiplication operations when compared to the Manhattan distance.

Preliminary results for k-means clustering indicate that speedups can be obtained using HPRC platforms, but additional research remains to complete the performance modeling, comparisons of performance on different RC and HPRC architectures, and to evaluate various scaling issues.

Image Segmentation

Segmentation is considered to be the first step in image analysis. The purpose of image segmentation is to subdivide an image into meaningful non-overlapping regions, which would be used for further analysis. The purpose of image segmentation is to decompose the image into parts that are meaningful with respect to a particular application. It is hoped that the regions obtained correspond to the physical parts or objects of a scene (3-D) represented by the image (2-D). One of the main purposes of image segmentation is to be used in Automatic Target Recognition (ATR) applications.

There are many segmentation algorithms available. The algorithm that is presented here is an image segmentation algorithm written in MATLAB by Dr. Allen Tannenbaum. The purpose of the segmentation algorithm is to segment Synthetic Aperture Radar (SAR) images. The SAR images are part of the public data set provided jointly by DARPA and Wright Laboratory as part of the Moving and Stationary Target Acquisition and Recognition (MSTAR) program. These are images of various military and synthetic targets taken from an airborne platform at various angles.

The program was designed to segment an image (specifically an image generated by a Synthetic Aperture Radar system) into different classes based on local statistics. The algorithm filters the image to get rid of small grain noise. Then it divides the features of the image into 3 different classifications: trees, grass, and shadows. It defines trees as having high mean and high variance, grass as having medium mean and high variance, and shadows as low mean and low variance. It will generate a new image with each pixel value replaced by the class number the algorithm determined that pixel to be drawn from. It replaces each pixel of the image with a classification code (1,2, or 3) indicating if it belongs to a tree, grass, or a shadow. This process produces a newly segmented image.

The data set for the project is taken from MSTAR data set. The Center for Imaging Science has been provided with this series of one-foot IPR synthetic aperture radar (SAR) images. This data was collected using the Sandia National Laboratories Twin Otter SAR sensor payload operating at X band. There are seventeen total image sets, with each set containing data for a particular target imaged at a particular depression angle. Each set contains data for hundreds of degrees of target aspect pose for that target at that angle of depression.

To support related research at the Oak Ridge National Laboratory, another application chosen for investigating the feasibility of using HPRC is “Digital Holographic

Reconstruction” to support semiconductor manufacturing. The “core” of this application is the FFT algorithm. It is an algorithm that is typically time-consuming even on fast computers. The FFT algorithm is being considered for different applications and for different research purposes. First, we are considering FFT performance on FPGAs, parallel processors, and HPRC platforms to compare the performance of each. Second, we are using the FFT algorithm to assess the effectiveness of design capture tools followed by the flow to map the algorithm to running executables on HPRC platforms.

Discrete-event simulation is an important application for the development, integration, validation, and assessment of systems. For electronic systems, as much as 70% of design time is spent in verification, and simulation represents the most commonly used technique. The Air Force’s emerging Joint Battlespace Infosphere development depends on JBISim to help in understanding the tradeoffs to be made in developing the most effective deployment. We are addressing discrete-event simulation with HPRC by exploiting the well-known parallel and distributed discrete-event simulation technology for the processing nodes, combined with customized acceleration of aspects of the simulation computation using FPGAs. For example, pseudo-random number generation according to specific probability distributions can be sped up, as can simulation execution units, state saving, GVT calculation, and fossil collection, communications support, and event queue support. We have developed a preliminary implementation of Petri net simulation using FPGAs, with a potential execution of 100MHz or higher. The Petri net simulator has hardware implementations to determine which transitions are enabled, selection circuitry to choose which transition fires based on a probability distribution, firing circuitry to consume and produce tokens as transitions fire, and other simulation infrastructure. Ongoing work with simulation will address design automation, parallel processing issues for very large simulations, and circuitry to better collect statistics and state. This preliminary work shows the promise for using HPRC to accelerate simulations.

Conclusions and Future Work

High Performance Reconfigurable Computing promises to provide significant performance improvements for a variety of applications. The coarse-grained parallelism that is traditionally exploited with parallel supercomputing/HPC platforms can be similarly used with HPRC. In addition, the fine-grained parallelism that reconfigurable computing devices (such as FPGAs) can effectively exploit provides additional performance benefit. Unfortunately, programming HPRC platforms faces the software development challenges of parallel and distributed programs combined with the need for hardware design expertise. We presented research focused on improving the programming infrastructure by developing an accurate performance model to help in understanding performance bottlenecks and in system optimization, assessing the best design tools and their effectiveness in capturing algorithms and ease in mapping to HPRC platforms, considering the most appropriate computational models for applications on HPRC, communications issues between the various reconfigurable logic and processing elements, and evaluating these consideration in the context of demonstration applications.

The preliminary performance modeling results are quite promising, with great potential for helping in developing or tuning applications on HPRC. We have completed initial work on model support for characterizing the hardware performance, communications costs, application load imbalance, background load imbalance, and effect of heterogeneous processing elements. Some validation work has been completed, and a number of applications are being completed to complete the validation work. Based on the results of these validation efforts, the model will be refined as needed. In addition, the application of this model to support application development will be pursued.

Preliminary work has begun on a representation of the communications capabilities of HPRC platforms, applications requirements, and a means of mapping the applications requirements to the HPRC platform. This work is still in its early stages, but looks quite promising as a theoretic framework for design automation on HPRC. It will also be used to extend the performance modeling work in progress. This approach uses a graph theory to describe the system and problem, and graph algorithms to help in the partitioning, mapping, and scheduling.

A number of applications have been developed using RC elements that help us to better understand the most appropriate techniques for using FPGAs. Similarly, a host of parallel and distributed applications have been developed and their performance characterized. Little has been done to characterize HPRC application needs or their performance. Our work with signal and image processing, graph theoretic algorithms, Boolean satisfiability, and simulation has helped to better understand design and performance issues and tradeoffs. We have discussed some of the techniques that are most appropriate for effectively employing HPRC platforms, but additional work needs to be performed to create a theoretic framework in which to discuss these issues. This framework is the subject of future work.

Acknowledgements

We would like to thank the Air Force Research Laboratory for supporting this research. In particular, thanks to Steve Drager for his support and input, as well as Rich Linderman, Bob Hillman, Jim Hanna, and Lt. Louis Pochet for their ideas and feedback. Don Bouldin and Mike Langston at The University of Tennessee provided valuable insight and feedback as well.

A number of graduate students are working on projects discussed in this report and helped with the results reported. They include Melissa Smith, Ashwin Balakrishnan, Venkatesh Bhaskaran, Siddhartha Devalapalli, Mahesh Dorai, Hongtau Du, Sampath Kothandaraman, Saumil Merchant, Bhanu Rekapalli, and Nitin Tiwari. In particular, this project helped support Melissa, Ashwin, Venkatesh, and Hongtau in their studies.

Additional support for this work came from The University of Tennessee's Center for Information Technology Research, the National Science Foundation, and Oak Ridge National Laboratory. We thank them as well for their support of our research.

References

- [1] Amdahl, G. M., Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, 30, pp. 483-485, 1967, Reston, VA,
- [2] Annapolis Microsystems, <http://www.annapmicro.com>, 2001,
- [3] Atallah, M. J., Black, C. L., Marinescu, D. C., Segel, H. J., and Casavant, T. L., "Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations," *Journal of Parallel and Distributed Computing*, vol. 16 pp. 319-327, 1992.
- [4] Roger D. Chamberlain, Mark A. Franklin, and Praveen Krishnamurthy, "[Optical Network Reconfiguration for Signal Processing Applications](#)." In *Proc. of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, July 2002, pp. 344-355.
- [5] R.D. Chamberlain, "Fast Fourier Transform on a Hypercube Architecture Augmented with a Synchronization Network." Tech. Rep. WUCCRC-93-16, June 1993.
- [6] R.D. Chamberlain, "Gaussian Elimination on a Hypercube Architecture Augmented with a Synchronization Network." Tech. Rep. WUCCRC-91-12, April 1991.
- [7] Clement, M. J. and Quinn, M. J., Analytical Performance Prediction on Multicomputers, *Proceedings of Supercomputing '93*, 1993,
- [8] DeHon, Andre, "Reconfigurable Architectures for General-Purpose Computing." Ph.D. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1996.
- [9] DeHon, A., "Comparing Computing Machines," *Proceedings of SPIE*, vol. 3526, no. Configurable Computing: Technology and Applications, pp. 124, Nov.1998.
- [10] Inderjit S.Dhillon, Dharmendra S. Modha, "A Data-Clustering Algorithm on Distributed Memory Multiprocessors", IBM Research Report RJ 10134(95009), Nov.11, 98, Proc Large-scale Parallel KDD Systems Workshop, ACM SIGKDD, Aug. 15-18, 99
- [11] Mike Estlick, Miriam Leiser, James Theiler, John J. Szymanski, "Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware"
- [12] Gokhale, M., Homes, W., Kopser, A., Lucas, S., Minnich, R., Sweely, D., and Lopresti, D., "Building and Using a Highly Parallel Programmable Logic Array," *IEEE Computer*, vol. 24, no. 1, pp. 81-89, Jan.1991.
- [13] Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R. R., "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, pp. 70-77, Apr.2000.
- [14] Gustafson, J. L., "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, pp. 532-533, May1988.

- [15] Hauck, S., Fry, T. W., Hosler, M. M., and Kao, J. P., "The Chimaera Reconfigurable Functional Unit," *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. -10, 1997.
- [16] Hauser, J. R. and Wawrzynek, J., "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.
- [17] Hu, L. and Gorton, I., Performance Evaluation for Parallel Systems: A Survey, UNSW-CSE-TR-9707, pp. -56, 1997, Sydney, Australia, University of NSW, School of Computer Science and Engineering,
- [18] I.S.I.East, SLAAC: System-Level Applications of Adaptive Computing, <http://>, 1998,
- [19] X. Jia. "Classification techniques for hyperspectral remote sensing image data", PhD thesis, Univ. College, ADFA, University of New South Wales, Australia, 1996.
- [20] Shailesh Kumar and Joydeep Ghosh and Melba M. Crawford, "Classification of Hyperspectral Data using Best-bases Feature Extraction", 2000
- [21] Edward A. Lee, "[Overview of the Ptolemy Project](#)," *Technical Memorandum UCB/ERL M01/11*, University of California, Berkeley, March 6, 2001.
- [22] E. A. Lee, "`[Overview of the Ptolemy Project](#)", ERL Technical Report UCB/ERL No. M98/71, University of California, Berkeley, CA 94720, November 23, 1998. *This paper has been superseded by Technical Memorandum UCB/ERL M01/11, March 6, 2001, also entitled [Overview of the Ptolemy Project](#)*
- [23] Edward A. Lee, "[Computing for Embedded Systems](#)," *IEEE Instrumentation and Measurement Technology Conference*, Budapest, Hungary, May 21-23, 2001.
- [24] E. A. Lee and A. Sangiovanni-Vincentelli, "`[A Framework for Comparing Models of Computation](#)," *IEEE Transactions on CAD*, Vol. 17, No. 12, December 1998.
- [25] Leong, P. H. W., Leong, M. P., Cheung, O. Y. H., Tung, T., Kwok, C. M., Wong, M. Y., and Lee, K. H., Pilchard - A Reconfigurable Computing Platform With Memory Slot Interface, Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001, California USA, IEEE,
- [26] Mohapatra, P., Das, C. R., and Feng, T., "Performance Analysis of Cluster-Based Multiprocessors," *IEEE Transactions on Computers*, pp. 109-115, 1994.
- [27] Moll, L., Vuillemin, J., and Boucard, P., High-Energy Physics on DECPeRLe-1 Programmable Active Memory, ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 47-52, 1995,
- [28] Myers, M., Jaget, K., Cadambi, S., Weener, J., Moe, M., Schmit, H., Goldstein, S. C., and Bowersox, D., PipeRench Manual, pp. -41, 1998, Carnegie Mellon University,

- [29] Nallatech FPGA-Centric Systems & Design Services, <http://www.nallatech.com/>, 2002,
- [30] NetSolve, <http://icl.cs.utk.edu/netsolve/>, 2001,
- [31] Noble, B. L. and Chamberlain, R. D., Performance Model for Speculative Simulation Using Predictive Optimism, Proceedings of the 32nd Hawaii International Conference on System Sciences, pp. 1-8, 1999,
- [32] Noble, B. L. and Chamberlain, R. D., Analytic Performance Model for Speculative, Synchronous, Discrete-Event Simulation, Proc.of 14th Workshop on Parallel and Distributed Simulation, 2000,
- [33] Cameron Patterson, "High Performance DES Encryption in Virtex FPGAs using JBits." *Proceedings of High Performance Embedded Computing Workshop*, Boston, MA, September 2000.
- [34] Peterson, G. D. and Smith, M. C., "Programming High Performance Reconfigurable Computers," *SSGRR 2001*, 2001, Rome, Italy.
- [35] Peterson, Gregory D., " Parallel Application Performance on Shared, Heterogeneous Workstations." Doctor of Science Washington University Sever Institute of Technology, Saint Louis, Missouri, 1994.
- [36] Peterson, G. D. and Chamberlain, R. D., "Parallel application performance in a shared resource environment," *Distributed Systems Engineering*, vol. 3 pp. 9-19, 1996.
- [37] <http://ptolemy.berkeley.edu>
- [38] Reynolds, P. F., Jr. and Pancerella, C. M., Hardware Support for Parallel Discrete Event Simulations, TR-92-08, 1992, Computer Science Dept.,
- [39] Reynolds, P. F., Jr., Pancerella, C. M., and Srinivasan, S., Making Parallel Simulations Go Fast, 1992 ACM Winter Simulation Conference, 1992,
- [40] Reynolds, P. F., Jr., Pancerella, C. M., and Srinivasan, S., "Design and Performance Analysis of Hardware Support for Parallel Simulations," *Journal of Parallel and Distributed Computing*, Aug.1993.
- [41] SInRG: Scalable Intracampus Research Grid, <http://www.cs.utk.edu/sinrg/index.html>, 2001,
- [42] Theiler, J., M. Leeser, M. Estlick, and J.J. Szymanski, "Design issues for hardware implementation of an algorithm for segmenting hyperspectral imagery", 2000.
- [43] Thomasian, A. and Bay, P. F., "Analytic Queueing Network Models for Parallel Processing of Task Systems," *IEEE Transactions on Computers*, vol. C-35, no. 12, pp. 1045-1054, Dec.1986.
- [44] Underwood, K. D., Sass, R. R., and Ligon, W. B., III, A Reconfigurable Extension to the Network Interface of Beowulf Clusters, Proc.of the 2001 IEEE International Conference on Cluster Computing, pp. -10, 2001, IEEE Computer Society,
- [45] Virtual Computer Corporation, <http://www.vcc.com/index.html>, 2002,

- [46] Vuillemin, J., Bertin, P., Roncin, D., Shand, M. , Touati, H., and Boucard, P., "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Transactions on VLSI Systems*, vol. 4, no. 1, pp. 56-69, Mar.1996.
- [47] Jeffrey Wolff, "Dance Implementation Of K-means Clustering", May 2000
- [48] Ye, Z. A., Moshovos, A., Hauck, S., and Banerjee, P., "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," 1998.

Appendix A: Firebird Host Source Code

```
/*
 *                      Annapolis Micro Systems Inc.
 *                      190 Admiral Cochrane
 *                      Suite 130
 *                      Annapolis, MD 21401
 */

/*
 * File      : satsolve.c      (from intr_ex.c)
 * Project   : SATSolve
 * Description: Boolean Satisfy solver
 * Author    : Melissa C. Smith
 * Date      : 08/05/02
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <ctype.h>

#include "ws.h"
#include "ws_shared.h"
#include "satsolve.h"
#ifdef _C2WILD_
#include "ws_c2wild.h"
#endif

/*
 * Function:  sat_ws_run
 * Description: SAT solver
 * Arguments:
 *   WS_Board      Slot or board number to access.
 *   dPEMask       PeMask to test.
 *   WS_Cfg        Configuration Information.
 * Returns:
 *   WS_SUCCESS    upon successful completion.
 */

WS_RetCode
sat_ws_run(DWORD WS_Board, DWORD dPEMask, WS_PhysicalBoardConfig *WS_Cfg)
{
```

```

WS_RetCode
    rc = WS_SUCCESS;

DWORD
    Reset,
    Data,
    pIntMask,
    dPE_Num,
    ValidIntrMask,
    IntToReset = 0x0,
    WildStar_IntStatus[3];

time_t
    start,
    finish;

Reset = 1;
Data = 1;
ValidIntrMask = 0x0;
WildStar_IntStatus[0] = 0x6;
WildStar_IntStatus[1] = 0x4;
WildStar_IntStatus[2] = 0x0;

/*****
**          Reset the PE          **
*****/
for ( dPE_Num = WS_PE0; dPE_Num <= WS_PE2; dPE_Num++ )
{
    if ( BitTst( dPEMask,dPE_Num ) )
    {
        printf( "  Resetting PE[%d] ... ", dPE_Num );
        rc = WS_WritePeReg( WS_Board, dPE_Num, INTR_RESET_BASE, 1, &Reset );
        CHECK_SUCCESS(rc);
        IntToReset = IntToReset | PENUM_TO_MASK(dPE_Num);
#ifdef _C2WILD_
        WS_Debug_Sleep(100);
#endif
        printf( "DONE\n" );
    }
}

/****

```

```

**          Clear all pending interrupts          **
*****/
rc = WS_ResetInterrupt( WS_Board, IntToReset );
CHECK_SUCCESS(rc);

printf ( "\n");

/*****/
**          **
** Test Pulsing of the interrupt lines          **
**          **
*****/
Data = 1;
printf( "PHASE 1: Pulse Interrupt Lines.\n" );
for ( dPE_Num = WS_PE0; dPE_Num <= WS_PE2; dPE_Num++ )
{
    if ( BitTst( dPEMask, dPE_Num ) )
    {

        ValidIntrMask = ((1 << dPE_Num) | ValidIntrMask);

        printf( " Setting and verifing interrupts on PE[%d] ... ", dPE_Num );
        /* Pulse Interrupt Line */
        rc = WS_WritePeReg( WS_Board, dPE_Num, INTR_PULSE_BASE, 1, &Data );
        CHECK_SUCCESS(rc);
        rc = WS_ReadPeReg( WS_Board, dPE_Num, INTR_PULSE_BASE, 1, &Data );
        CHECK_SUCCESS(rc);

        time(&start);
        do
        {
            time(&finish);
            if ( difftime(finish, start) >= (double)INTR_TIMEOUT_VALUE )
            {
                printf( "Failed to receive interrupt from PE[%d]\n", dPE_Num );
                rc = WS_FAILED_TO_RECEIVE_INTERRUPT;
                return(rc);
            }
        }
        else
        {
            /* Verify Interrupt Was received */
            rc = WS_QueryInterruptStatus( WS_Board, &pIntMask );
            CHECK_SUCCESS(rc);

```

```

    }
    }while (( rc == WS_SUCCESS) && ( pIntMask != ValidIntrMask ));

    printf( "Success\n");
}
}

/*****
**
**      Clear all interrupts & Verify no
**      interrupts are pending
**
**
*****/
printf( "\n");
printf( "Clearing and verifying interrupts ... ");
for ( dPE_Num = WS_PE0; dPE_Num <= WS_PE2; dPE_Num++ )
{
    if ( BitTst( dPEMask, dPE_Num ) )
    {

        rc = WS_ResetInterrupt( WS_Board, PENUM_TO_MASK(dPE_Num) );
        CHECK_SUCCESS(rc);

        /* Verify there are no interrupts pending */
        rc = WS_QueryInterruptStatus( WS_Board, &pIntMask );
        CHECK_SUCCESS(rc);

        if ( dPEMask == WS_PE1_MASK )
        {
            if ( pIntMask != 0x0 )
            {
                printf( "Interrupt not cleared on PE[%d]\n", dPE_Num );
                rc = WS_FAILED_TO_RECEIVE_INTERRUPT;
                return(rc);
            }
        }
        else
        {
            if ( pIntMask != WildStar_IntStatus[dPE_Num])
            {
                printf( "Interrupt not cleared on PE[%d]\n", dPE_Num );
                rc = WS_FAILED_TO_RECEIVE_INTERRUPT;
                return(rc);
            }
        }
    }
}

```

```

    }
}
}
}
printf( "Done.\n" );
printf( "\n");

/*****
**
**      Test Query interrupt status API      **
**
**
*****/
Data = 0x1;
ValidIntrMask = 0x0;
printf( "PHASE 2: Test WS_WaitOnInterrupt API.\n" );

for ( dPE_Num = WS_PE0; dPE_Num <= WS_PE2; dPE_Num++ )
{
    if ( BitTst( dPEMask, dPE_Num ) )
    {

        printf( "    Testing PE[%d]\n", dPE_Num );

        /* Start counter */
        printf( "    Starting Counter ...      " );
        rc = WS_WritePeReg( WS_Board, dPE_Num, INTR_COUNTER_BASE, 1, &Data );
        CHECK_SUCCESS(rc);
        printf( "Done\n" );

        printf( "    Waiting for interrupt ... " );
        rc = WS_WaitOnInterrupt ( WS_Board, PENUM_TO_MASK(dPE_Num), &pIntMask,
INTR_TIMEOUT_VALUE );
        CHECK_SUCCESS(rc);
        printf( "Done\n" );

        printf( "    Verifying Mask ...      " );
        /* Verify only the newest interrupt is in the mask returned from the
WS_WaitOnInterrupt call */
        if ( pIntMask != ( 1 << dPE_Num ) )
        {
            printf( "    Interrupt Mask Incorrect\n");
            printf( "    Expected 0x%02x    Received 0x%02x \n", 1 << dPE_Num, pIntMask
);
            rc = ERROR_TEST_FAILED;

```

```

        return(rc);
    }

    /* Verify all interrupts are still there because we have not cleared them
*/
    rc = WS_QueryInterruptStatus( WS_Board, &pIntMask );
    CHECK_SUCCESS(rc);

    ValidIntrMask = ((1 << dPE_Num) | ValidIntrMask);
    if ( pIntMask != ValidIntrMask )
    {
        printf( " Cumulative Interrupt Mask Incorrect\n");
        printf( "      Expected 0x%02x      Received 0x%02x \n", ValidIntrMask,
pIntMask );
        rc = ERROR_TEST_FAILED;
        return(rc);
    }
    printf( "Done\n" );
    printf( "\n");
}
}

/*****
**   Reset the PE   to clear PE's request   **
*****/
for ( dPE_Num = WS_PE0; dPE_Num <= WS_PE2; dPE_Num++ )
{
    if ( BitTst( dPEMask,dPE_Num ) )
    {
        Reset = 0x1;
        printf( "   Resetting PE[%d] ... ", dPE_Num );
        rc = WS_WritePeReg( WS_Board, dPE_Num, INTR_RESET_BASE, 1, &Reset );
        CHECK_SUCCESS(rc);
        printf( "DONE\n" );
    }
}

/*****
**           Clear all pending interrupts           **
*****/
rc = WS_ResetInterrupt( WS_Board, IntToReset );
CHECK_SUCCESS(rc);

return (rc);

```



```

}

/*****
* Function:  sat_fb_run
* Description:  Sat solver
* Arguments:
*   WS_Board      Slot or board number to access.
*   dPEMask       PeMask to test.
*   WS_Cfg        Configuration Information.
* Returns:
*   WS_SUCCESS    upon successful completion.
*****/
WS_RetCode
sat_fb_run(DWORD WS_Board, DWORD dPEMask, WS_PhysicalBoardConfig *WS_Cfg)
{
    WS_RetCode
        rc = WS_SUCCESS;

    DWORD
        pIntMask,
        dPE_Num,
        ValidIntrMask,
        IntToReset = 0x0,
        WildStar_IntStatus[3];

    DWORD
        Reset,
        Data;

    time_t
        start,
        finish;

    Reset = 1;
    Data = 1;
    ValidIntrMask = 0x0;
    WildStar_IntStatus[0] = 0x6;
    WildStar_IntStatus[1] = 0x4;
    WildStar_IntStatus[2] = 0x0;
    dPE_Num = WS_PE0;

    /*****
    **                      Reset the PE                      **
    *****/

```

```

*****/
printf( "   Resetting PE[%d] ... ", dPE_Num );
rc = WS_WritePeReg( WS_Board, WS_PE0, INTR_RESET_BASE<<1, 1, &Reset );
CHECK_SUCCESS(rc);
IntToReset = IntToReset | PENUM_TO_MASK(dPE_Num);
#ifdef _C2WILD_
    WS_Debug_Sleep(100);
#endif
printf( "DONE\n" );

/*****
**          Clear all pending interrupts          **
*****/
rc = WS_ResetInterrupt( WS_Board, IntToReset );
CHECK_SUCCESS(rc);

printf ( "\n");

/*****
**                                     **
**   Test Pulsing of the interrupt lines   **
**                                     **
*****/
Data = 1;
printf( "PHASE 1: Pulse Interrupt Lines.\n" );

ValidIntrMask = ((1 << dPE_Num) | ValidIntrMask);

printf( "   Setting and verifying interrupts on PE[%d] ... ", dPE_Num );
/* Pulse Interrupt Line */
rc = WS_WritePeReg( WS_Board, dPE_Num, INTR_PULSE_BASE<<1, 1, &Data );
CHECK_SUCCESS(rc);
rc = WS_ReadPeReg( WS_Board, dPE_Num, INTR_PULSE_BASE<<1, 1, &Data );
CHECK_SUCCESS(rc);

time(&start);
do
{
    time(&finish);
    if ( difftime(finish, start) >= (double)INTR_TIMEOUT_VALUE )
    {
        printf( "Failed to receive interrupt from PE[%d]\n", dPE_Num );
        rc = WS_FAILED_TO_RECEIVE_INTERRUPT;
    }
}

```

```

        return(rc);
    }
    else
    {
        /* Verify Interrupt Was received */
        rc = WS_QueryInterruptStatus( WS_Board, &pIntMask );
        CHECK_SUCCESS(rc);
    }
}while ( ( rc == WS_SUCCESS) && ( pIntMask != ValidIntrMask ));

printf( "Success\n");

/*****
**                                     **
**      Clear all interrupts & Verify no      **
**      interrupts are pending                **
**                                     **
*****/
printf( "\n");
printf( "Clearing and verifying interrupts ... ");

rc = WS_ResetInterrupt( WS_Board, PENUM_TO_MASK(dPE_Num) );
CHECK_SUCCESS(rc);

/* Verify there are no interrupts pending */
rc = WS_QueryInterruptStatus( WS_Board, &pIntMask );
CHECK_SUCCESS(rc);

if ( pIntMask != 0x0 )
{
    printf( "Interrupt not cleared on PE[%d]\n", dPE_Num );
    rc = WS_FAILED_TO_RECEIVE_INTERRUPT;
    return(rc);
}
printf( "Done.\n" );
printf( "\n");

/*****
**                                     **
**      Test Query interrupt status API      **
**                                     **
*****/
Data = 0x1;

```

```

ValidIntrMask = 0x0;
printf( "PHASE 2: Test WS_WaitOnInterrupt API.\n" );

printf( "   Testing PE[%d]\n", dPE_Num );

/* Start counter */
printf( "   Starting Counter ...   " );
rc = WS_WritePeReg( WS_Board, dPE_Num, INTR_COUNTER_BASE<<1, 1, &Data );
CHECK_SUCCESS(rc);
printf( "Done\n" );

printf( "   Waiting for interrupt ... " );
rc = WS_WaitOnInterrupt ( WS_Board, PENUM_TO_MASK(dPE_Num), &pIntMask,
INTR_TIMEOUT_VALUE );
CHECK_SUCCESS(rc);
printf( "Done\n" );

printf( "   Verifying Mask ...   " );
/* Verify only the newest interrupt is in the mask returned from the
WS_WaitOnInterrupt call */
if ( pIntMask != ( 1 << dPE_Num ) )
{
    printf( " Interrupt Mask Incorrect\n");
    printf( " Expected 0x%02x Received 0x%02x \n", 1 << dPE_Num, pIntMask );
    rc = ERROR_TEST_FAILED;
    return(rc);
}

/* Verify all interrupts are still there because we have not cleared them */
rc = WS_QueryInterruptStatus( WS_Board, &pIntMask );
CHECK_SUCCESS(rc);

ValidIntrMask = ((1 << dPE_Num) | ValidIntrMask);
if ( pIntMask != ValidIntrMask )
{
    printf( " Cumulative Interrupt Mask Incorrect\n");
    printf( " Expected 0x%02x Received 0x%02x \n", ValidIntrMask, pIntMask );
    rc = ERROR_TEST_FAILED;
    return(rc);
}
printf( "Done\n" );
printf( "\n");

/*****

```

```

**   Reset the PE   to clear PE's request   **
*****/

Reset = 0x1;
printf( "   Resetting PE[%d] ... ", dPE_Num );
rc = WS_WritePeReg( WS_Board, dPE_Num, INTR_RESET_BASE<<1, 1, &Reset );
CHECK_SUCCESS(rc);
printf( "DONE\n" );

/*****
**       Clear all pending interrupts       **
*****/
rc = WS_ResetInterrupt( WS_Board, IntToReset );
CHECK_SUCCESS(rc);

return (rc);
}

/*****
* Function:      satsolve_Startup
* Description:   Initializes the WILDSTAR board.
* Arguments:
*   WS_Board      Slot or board number to access.
*   dMaxIterations Maximum number of iterations.
*   dLocalBusSpeed Local Address Bus Speed.
*   dPEMask       PeMask to test.
*   *WS_Cfg       Configuration Information.
* Returns:
*   WS_SUCCESS    upon successful completion, or
*   rc            Return code of failure.
*****/
WS_RetCode
satsolve_startup(DWORD WS_Board, DWORD dMaxIterations, DWORD dLocalBusSpeed,
DWORD dPEMask, WS_PhysicalBoardConfig *WS_Cfg)
{
    WS_RetCode
        rc = WS_SUCCESS;

    char
        PathName[MAX_PATH],
        cAppend[4],
        cPEName[7];

#ifdef WSCPU_MVME2604 || defined(WSCPU_I960)

```

```

USHORT
    iLevel,
    iVector;

/*****
**
**          Get the VME interrupt level/vector
**
**          *****/
rc = WS_GetVmeInterruptVector( WS_Board, &iLevel, &iVector );
printf("  Current VME interrupt level/vector: %d/0x%x\n", iLevel, iVector );
CHECK_RC(rc);

/*****
**
**          Set the VME interrupt level/vector
**
**          *****/
if ( iVector == 0xFE )
{
    iVector = WS_DEFAULT_INTR_VECTOR;
}
printf("  Set VME interrupt level/vector: %d/0x%x\n", WS_DEFAULT_INTR_LEVEL,
iVector );
rc = WS_SetVmeInterruptVector( WS_Board,
                               WS_DEFAULT_INTR_LEVEL,
                               iVector );

CHECK_RC(rc);
#endif

if ( WS_Cfg->BaseInfo.BaseBoardType == WS_FIREBIRD_PMC )
{
    if ( IS_FIREBIRD_PMC_WSDP(WS_Cfg) ) /* Card is a Firebird_WSDP */
    {
        sprintf ( cAppend, "_ws");
    }
    else if ( IS_FIREBIRD_PMC_GLINK(WS_Cfg) ) /* Card is a Firebird_GLink */
    {
        sprintf ( cAppend, "_gl");
    }
}
else if ( WS_Cfg->BaseInfo.BaseBoardType == WS_FIREBIRD_PCI )
{

```

```

        cAppend[0] = '\\0';
    }
else
{
    if ( dLocalBusSpeed <= 0x3)
    {
        sprintf ( cAppend, "_33" );
    }
    else
    {
        sprintf ( cAppend, "_66" );
    }
}

/*****
**                               Set MClk clock                               **
*****/
rc = WS_MClkSetConfig ( WS_Board, PROG_OSCILLATOR, 30.0, 1 );
CHECK_SUCCESS(rc);
printf( "Successfully Set MCLK to [%2.1f]\\n", 30.0 );

#ifdef _C2WILD_
    return(rc);
#endif

/*****
**                               Program PE0                               **
*****/
if ( BitTst( dPEMask,WS_PE0) )
{
    sprintf ( cPEName, "pe0%s", cAppend );
    rc = GetPeImagePath( WS_BASEBOARD, WS_PE_DEVICE_ID, WS_PE0, WS_Cfg, cPEName,
        PathName );
    CHECK_SUCCESS(rc);

    printf("Programming PE0 with file %s ...", PathName);
    rc = ProgramPeFromFile( WS_Board, WS_PE0, PathName );
    CHECK_SUCCESS(rc);

    printf ( "Successful\\n\\n" );
}

/*****
**                               Program PE1                               **
*****/

```

```

*****/
if ( BitTst( dPEMask,WS_PE1) )
{
    sprintf ( cPEName, "pex%s", cAppend );
    rc = GetPeImagePath( WS_BASEBOARD, WS_PE_DEVICE_ID, WS_PE1, WS_Cfg, cPEName,
PathName );
    CHECK_SUCCESS(rc);

    printf("Programming PE1 with file %s ...", PathName);
    rc = ProgramPeFromFile( WS_Board, WS_PE1, PathName );
    CHECK_SUCCESS(rc);

    printf ( "Successful\n\n" );
}

/*****
**                      Program PE2                      **
*****/
if ( BitTst( dPEMask,WS_PE2) )
{
    sprintf ( cPEName, "pex%s", cAppend );
    rc = GetPeImagePath( WS_BASEBOARD, WS_PE_DEVICE_ID, WS_PE2, WS_Cfg, cPEName,
PathName );
    CHECK_SUCCESS(rc);

    printf("Programming PE2 with file %s ...", PathName);
    rc = ProgramPeFromFile( WS_Board, WS_PE2, PathName );
    CHECK_SUCCESS(rc);

    printf ( "Successful\n\n" );
}
return (rc);
}

/*****
* Function:          satsolve_Shutdown
* Description:       Loads "safe" PE images to put the board in a know state.
* Arguments:
*   WS_Board         Slot or board number to access.
*   dPEMask          PeMask to test.
*   *WS_Cfg          Configuration Information.
* Returns:
*   WS_SUCCESS       upon successful completion, or
*   rc               Return code of failure.
*****/

```



```

***** /
WS_RetCode
satsolve_shutdown(DWORD WS_Board, DWORD dPEMask, WS_PhysicalBoardConfig *WS_Cfg)
{
    WS_RetCode
    rc = WS_SUCCESS;

    printf ( "\nDeProgramming:\n");

    /*****
    **                      DeProgram PE0                      **
    *****/
    if ( BitTst( dPEMask,WS_PE0) )
    {
        printf ( "\tWS_PE0 ... ");
        rc = WS_DeProgramPe( WS_Board, WS_PE0_MASK);
        CHECK_SUCCESS(rc);

        printf ( "Successful\n\n" );
    }

    /*****
    **                      DeProgram PE1                      **
    *****/
    if ( BitTst( dPEMask,WS_PE1) )
    {
        printf ( "\tWS_PE1 ... ");
        rc = WS_DeProgramPe( WS_Board, WS_PE1_MASK);
        CHECK_SUCCESS(rc);

        printf ( "Successful\n\n" );
    }

    /*****
    **                      DeProgram PE2                      **
    *****/
    if ( BitTst( dPEMask,WS_PE2) )
    {
        printf ( "\tWS_PE2 ... ");
        rc = WS_DeProgramPe( WS_Board, WS_PE2_MASK);
        CHECK_SUCCESS(rc);

        printf ( "Successful\n\n" );
    }
}

```

```

    }

    return (rc);
}

/*****
* Function:  satsolve_ex
* Description:    SAT Solver with Interrupt test
* Arguments:
*   WS_Board      Slot or board number to access.
*   dPEMask       PeMask to test.
* Returns:
*   WS_SUCCESS    upon successful completion.
*****/
WS_RetCode
satsolve_ex(DWORD WS_Board, DWORD dMaxIterations, DWORD dLocalBusSpeed, DWORD
dPEMask)
{
    WS_RetCode
        rc = WS_SUCCESS;

    DWORD
        i,
        dSuccess,
        dFailure,
        dIteration,
        pVersion[4],
        pdVersion[8];

    static WS_PhysicalBoardConfig
        WS_Cfg;

    dSuccess = 0;
    dFailure = 0;

    /*****
    **           Get board configuration information           **
    *****/
    rc = WS_GetPhysicalConfig( WS_Board, &WS_Cfg );
    CHECK_SUCCESS(rc);

    if ( WS_Cfg.BaseInfo.BaseBoardType == WS_STARFIRE )
    {
        dPEMask = dPEMask & 0x2;
    }

```

```

else if ( ( WS_Cfg.BaseInfo.BaseBoardType == WS_WILDSTAR_PCI ) ||
          ( WS_Cfg.BaseInfo.BaseBoardType == WS_WILDSTAR_VME ) ||
          ( WS_Cfg.BaseInfo.BaseBoardType == WS_WILDSTAR_PCI_E ) ||
          ( WS_Cfg.BaseInfo.BaseBoardType == WS_WILDSTAR_VME_E ) )
{
    dPEMask = dPEMask & 0x7;
}
else if ( ( WS_Cfg.BaseInfo.BaseBoardType == WS_FIREBIRD_PCI ) ||
          ( WS_Cfg.BaseInfo.BaseBoardType == WS_FIREBIRD_PMC ) )
{
    dPEMask = dPEMask & 0x1;
}

#ifdef _C2WILD_
    WS_Debug_Sleep(200);
#endif

/*****
**          Display configuration information          **
*****/
#ifdef _C2WILD_
    rc = DisplayConfiguration( WS_Board, &WS_Cfg );
    CHECK_SUCCESS(rc);
#endif

/*****
**          Setup clocks and program pe(s)          **
*****/
rc = satsolve_startup(WS_Board, dMaxIterations, dLocalBusSpeed, dPEMask,
&WS_Cfg);
CHECK_SUCCESS(rc);

printf( "Example setup successful\n\n" );

#ifdef _C2WILD_
    WS_Debug_Sleep(200);
#endif

/*****
**          Display information about the test          **
*****/
printf("[Version Info]\n\n" );

rc=WS_DisplayIntensity( WS_Board, Level_2 );
CHECK_SUCCESS(rc);

```

```

rc = WS_UpdateDisplay( WS_Board, "INTR" );
CHECK_SUCCESS(rc);

if ( ( WS_Cfg.BaseInfo.BaseBoardType == WS_FIREBIRD_PCI ) ||
    ( WS_Cfg.BaseInfo.BaseBoardType == WS_FIREBIRD_PMC ) )
{
    rc = WS_ReadPeReg(WS_Board, WS_PE0, (INTR_VERSION_BASE<<1), 8, pdVersion);
    for ( i = 0; i <= 3;i++ )
        pVersion[i] = pdVersion[2*i];
}
else
{
    rc = WS_ReadPeReg(WS_Board, WS_PE1, INTR_VERSION_BASE, 4, pVersion);
}
CHECK_SUCCESS(rc);

printf("Version Register\n");
printf("  VHDL Version          : %d.%d\n",
    (pVersion[VHDL_VERSION_INDEX] >> VHDL_MAJOR_VERSION_SHIFT) &
VERSION_MASK,
    (pVersion[VHDL_VERSION_INDEX] >> VHDL_MINOR_VERSION_SHIFT) &
VERSION_MASK);
if (pVersion[VHDL_VERSION_INDEX] & IO_VHDL_MASK) /*Print IO VHDL version only
if needed*/
{
    printf("  IO VHDL Version      : %d.%d\n",
        (pVersion[VHDL_VERSION_INDEX] >> IO_VHDL_MAJOR_VERSION_SHIFT) &
VERSION_MASK,
        (pVersion[VHDL_VERSION_INDEX] >> IO_VHDL_MINOR_VERSION_SHIFT) &
VERSION_MASK);
}
printf("  ModelTech Version      : %d.%d%-2X\n",
    (pVersion[MTECH_VERSION_INDEX] >> MAJOR_VERSION_SHIFT) & VERSION_MASK,
    (pVersion[MTECH_VERSION_INDEX] >> MINOR_VERSION_SHIFT) & VERSION_MASK,
    (pVersion[MTECH_VERSION_INDEX] >> REV_VERSION_SHIFT) & VERSION_MASK);
printf("  Synpilfy Version       : %d.%d.%d\n",
    (pVersion[SYN_VERSION_INDEX] >> MAJOR_VERSION_SHIFT) & VERSION_MASK,
    (pVersion[SYN_VERSION_INDEX] >> MINOR_VERSION_SHIFT) & VERSION_MASK,
    (pVersion[SYN_VERSION_INDEX] >> REV_VERSION_SHIFT) & VERSION_MASK);
printf("  Xilinx Version         : %d.%di SP %d\n\n",
    (pVersion[XIL_VERSION_INDEX] >> MAJOR_VERSION_SHIFT) & VERSION_MASK,
    (pVersion[XIL_VERSION_INDEX] >> MINOR_VERSION_SHIFT) & VERSION_MASK,
    (pVersion[XIL_VERSION_INDEX] >> REV_VERSION_SHIFT) & VERSION_MASK);

```

```

printf("[SATsolve_ex]\n\n" );

/*****
**                               **
**               Run the test               **
**                               **
*****/
for ( dIteration = 1; dIteration <= dMaxIterations; dIteration ++ )
{
    printf ( "Iteration [%d] of [%d]\n", dIteration , dMaxIterations);
    if ( ( WS_Cfg.BaseInfo.BaseBoardType == WS_FIREBIRD_PCI ) ||
        ( WS_Cfg.BaseInfo.BaseBoardType == WS_FIREBIRD_PMC ) )
    {
        rc = sat_fb_run(WS_Board, dPEMask, &WS_Cfg);
        // test interrupt with feedback
    }
    else
    {
        rc = sat_ws_run(WS_Board, dPEMask, &WS_Cfg);
        // enable SATsolver and wait for interrupt
    }
    if ( rc == WS_SUCCESS)
    {
        dSuccess++;
    }
    else
    {
        dFailure++;
    }
    printf ("\n\t**** Successful[%d] Failures[%d] ****\n", dSuccess, dFailure);
}

/*****
**                               **
**               Leave the board in a known state               **
**                               **
*****/
rc = satsolve_shutdown(WS_Board, dPEMask, &WS_Cfg);
CHECK_SUCCESS(rc);

printf( "Example shutdown successful\n\n" );

return (rc);
}

/*****
*                               *
*               Main               *
*                               *
*****/

```

```

#if defined(WSCPU_MVME2604) || defined(WSCPU_I960)
WS_RetCode intr_ex( DWORD WS_Board, DWORD dLocalBusSpeed )
{
#else
int
main( int argc, char *argv [] )
{
    int
        argi;

    DWORD
        WS_Board = DEFAULT_BOARD,
        dLocalBusSpeed = DEFAULT_LOCAL_BUS_SPEED;
#endif

    WS_RetCode
        rc = WS_SUCCESS;

    DWORD
        dMaxIterations,
        dPEMask;

    const char * help_string =
        "Usage: intr_ex <list of options>\n"
        "  Options:\n"
        "    -I <Iterations>  Set the number of iterations.    ( default = 1 )\n"
        "    -M <Mask>         Set the mask of the PE to test.   ( default = 0x7 )\n"
        "    -L <Speed>        Set the Local LAD Bus Speed.     ( default = 33 )\n"
        "    -b <Board>        Set the Wildstar(tm) Board Number. ( default = 0 )\n";

    dMaxIterations = DEFAULT_ITERATIONS;
    dPEMask = DEFAULT_PEMASK;

#if !defined(WSCPU_MVME2604) && !defined(WSCPU_I960)
    for ( argi = 1; argi < argc; argi++ )
    {
        if ( argv [ argi ] [ 0 ] == '-' )
        {
            switch ( toupper( argv [ argi ] [ 1 ] ) )
            {
                case 'B': /* Set the Wildstar(tm) Board Number */
                    argi++;
                    if (argi < argc)

```

```

{
    WS_Board = strtoul( argv [ argi ], NULL, 16 );
    printf("Setting Board Number to %x\n", WS_Board );
}
else
{
    printf( " Warning: Invalid Board Number!\n");
    return(0);
}
if ( ( WS_Board < 0 ) || ( WS_Board > WS_MAX_BOARDS ) )
{
    printf( " Warning: Invalid Board/Slot Number!\n");
    printf ( "%s\n\n", help_string );
    return(0);
}
break;

case 'I': /* Select number of Loops */
    argi++;
    if (argi < argc)
    {
        dMaxIterations = atoi( argv [ argi ] );
    }
    else
    {
        printf( " Warning: Invalid loop option\n" );
        printf ( "%s\n\n", help_string );
        return(0);
    }
    break;

case 'M': /* Set the PE mask */
    argi++;
    if (argi < argc)
    {
        dPEMask = strtoul( argv [ argi ], NULL, 0 );
        printf( "Setting the PE Mask to %d.\n", dPEMask);
    }
    else
    {
        dPEMask = 0x7;
        printf("Warning: Invalid PE Mask ( Setting PEMask to %X )\n", 0x7);
    }
}

```

```

        break;

    case 'L': /* Set Local Address/Data bus speed */
        argi++;
        if (argi < argc)
        {
            dLocalBusSpeed = strtoul( argv [ argi ], NULL, 10 );
        }
        break;

    default:
        printf ( "   Unknown option: \"%s\"\n", argv [ argi ] );
        printf ( "%s\n\n", help_string );
        return(0);
    }
}
}
#endif

if ( dLocalBusSpeed == 66 )
{
    dLocalBusSpeed = WS_66MHZ_LADBUS_FLAG | WS_SHARED_FLAG;
}
else if ( dLocalBusSpeed == 33 )
{
    dLocalBusSpeed = WS_33MHZ_LADBUS_FLAG | WS_SHARED_FLAG;
}
else
{
    printf( "Invalid Bus Speed option\n" );
    printf ( "%s\n\n", help_string );
    return(0);
}

/*****
**           Open the board for testing           **
*****/
rc = WS_Open( WS_Board, dLocalBusSpeed );
CHECK_SUCCESS(rc);

/*****
**           Run example           **
*****/

```



```

*****/
rc = satsolve_ex(WS_Board, dMaxIterations, dLocalBusSpeed, dPEMask);
CHECK_SUCCESS(rc);

/*****
**                Close the board                **
*****/
rc = WS_Close( WS_Board );
CHECK_SUCCESS(rc);

return(0);
}

```

Appendix B: MATLAB Makefile Code

Matlab inherently does not support parallelism due to corporate decisions. But a lot of effort has been put into parallelizing Matlab codes at various levels of abstraction.

There are mainly 4 approaches to providing parallel functionalities to Matlab:

1. Provide communication routines (MPI/PVM) in Matlab.
2. Provide routines to split up work among multiple Matlab sessions.
3. Provide parallel backend to Matlab.
4. Compile Matlab scripts into native parallel code.

Our approach to exploit parallelism and to support RC APIs is to provide communication routines (MPI/PVM) in Matlab with the help of Matlab's External API interface.

Matlab inherently cannot support parallelism due to its basic architectural constraints. Hence by using the external API interface of Matlab, candidate portions of application code can be run in languages like C and Fortran which can very well take exploit the potential advantage of HPC and RC architectures. To exploit parallelism, MPI and PVM libraries are used in conjunction with C/C++ code for candidate portions of Matlab Code.

There are two approaches to consider in this, both having its pros and cons.

1. To use Matlab as the Master application that controls and manages the rest of the code and provide capability for C/C++ functions to be called from within Matlab Applications which in turn can exploit the parallelism by invoking the parallel libraries like MPI/PVM. It can also map the code down to the FPGAs for achieving higher performance by using RC boards.
2. The other is to keep the controllability and manageability in C/C++ code, which acts as a Master program invoking Matlab Engine. Again the C/C++ code can also map the candidate application code down to the FPGAs and also in turn use the MPI/PVM libraries to exploit parallelism.

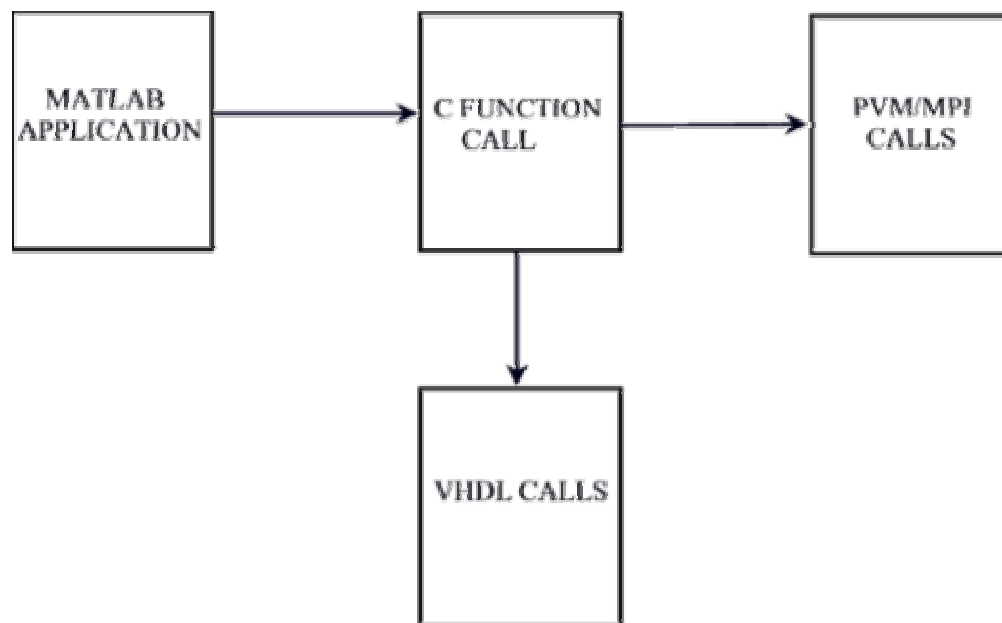


Fig 2. Framework A, Using Matlab Application as a master invoking C/C++ code.

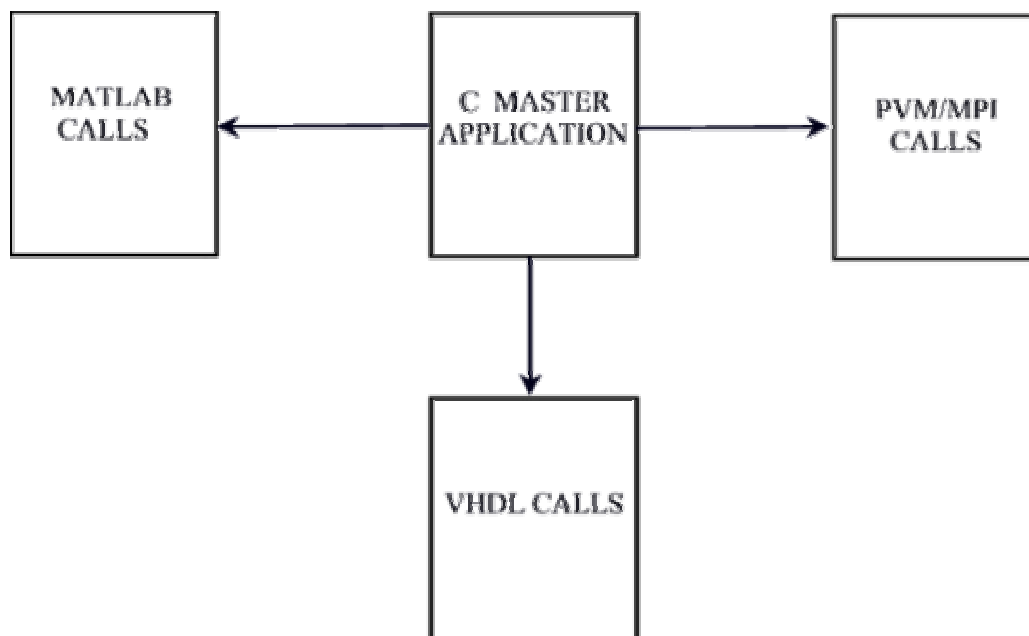


Fig 3. Framework B, Using C/C++ as a master code invoking Matlab Engine routines.

C subroutines can be called from MATLAB as if they were built-in functions. MATLAB callable C programs are referred to as MEX-files. MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute. The use of Mex files has several advantages like using the large pre-existing C programs that can be called from MATLAB without having to be rewritten as M-files and also bottleneck computations (usually for-loops) that do not run fast enough in MATLAB can be recoded in C for efficiency.

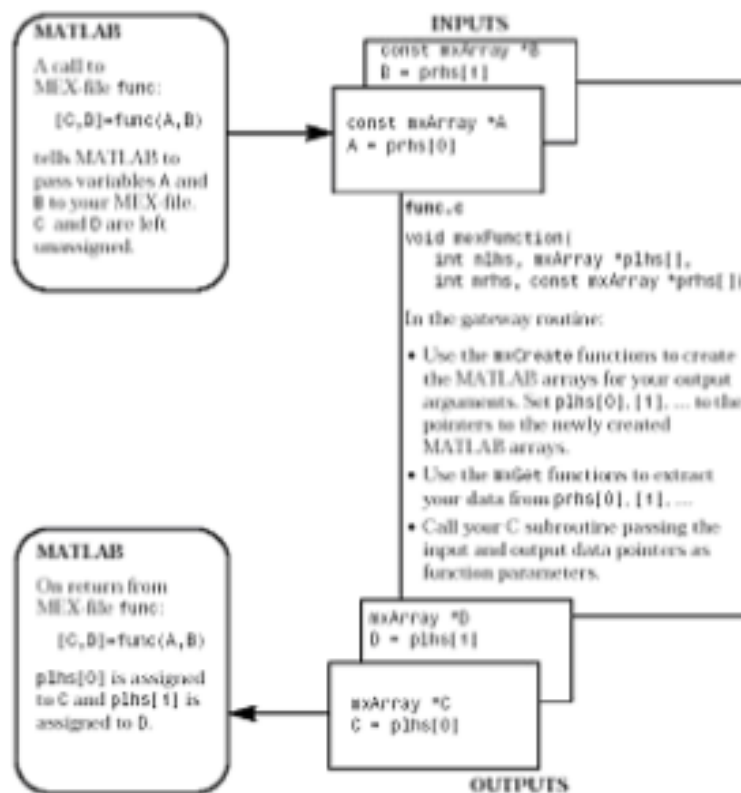


Fig 4. Flow chart explaining the method to use the Mex files

Figure 4 is a basic structure and a flow chart from Matlab External API interface documentation explaining the method to use Mex files.

C MEX-files are built by using the mex script to compile your C source code with additional calls to API routines. The PVM/MPI libraries used need to be linked.

The source code for a MEX-file consists of two distinct parts:

- A *computational routine* that contains the code for performing the computations that you want implemented in the MEX-file. Computations can be numerical computations as well as inputting and outputting data.
- A *gateway routine* that interfaces the computational routine with MATLAB by the entry point mexFunction and its parameters prhs, nrhs, plhs, nlhs, where prhs is an array of right-hand input arguments, nrhs is the number of right-hand input arguments, plhs is an array of left-hand output arguments, and nlhs is the number of left-hand output arguments. The gateway calls the computational routine as a subroutine.

The following demo C Program from Matlab documentation clarifies this approach.

Example C Program:

```

/*=====
 *
 * YPRIME.C Sample .MEX file corresponding to YPRIME.M
 *          Solves simple 3 body orbit problem
 *
 * The calling syntax is:
 *
 *          [yp] = yprime(t, y)
 *
 * You may also want to look at the corresponding M-code, yprime.m.
 *
 * This is a MEX-file for MATLAB.
 * Copyright 1984-2000 The MathWorks, Inc.
 *
 *=====*/
/* $Revision: 1.10 $ */
#include <math.h>
#include "mex.h"

/* Input Arguments */
#define T_IN prhs[0]
#define Y_IN prhs[1]

/* Output Arguments */
#define YP_OUT plhs[0]

#if !defined(MAX)
#define MAX(A, B) ((A) > (B) ? (A) : (B))
#endif

#if !defined(MIN)
#define MIN(A, B) ((A) < (B) ? (A) : (B))
#endif

#define PI 3.14159265

static double mu = 1/82.45;
static double mus = 1 - 1/82.45;

static void yprime(

```

```

        double    yp[],
        double    *t,
        double    y[]
    )
{
    double    r1,r2;

    r1 = sqrt((y[0]+mu)*(y[0]+mu) + y[2]*y[2]);
    r2 = sqrt((y[0]-mus)*(y[0]-mus) + y[2]*y[2]);

    /* Print warning if dividing by zero. */
    if (r1 == 0.0 || r2 == 0.0 ){
        mexWarnMsgTxt("Division by zero!\n");
    }

    yp[0] = y[1];
    yp[1] = 2*y[3]+y[0]-mus*(y[0]+mu)/(r1*r1*r1)-mu*(y[0]-
mus)/(r2*r2*r2);
    yp[2] = y[3];
    yp[3] = -2*y[1] + y[2] - mus*y[2]/(r1*r1*r1) - mu*y[2]/(r2*r2*r2);
    return;
}

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray*prhs[] )
{
    double *yp;
    double *t,*y;
    unsigned int m,n;

    /* Check for proper number of arguments */

    if (nrhs != 2) {
        mexErrMsgTxt("Two input arguments required.");
    } else if (nlhs > 1) {
        mexErrMsgTxt("Too many output arguments.");
    }

    /* Check the dimensions of Y.  Y can be 4 X 1 or 1 X 4. */

    m = mxGetM(Y_IN);
    n = mxGetN(Y_IN);
    if (!mxIsDouble(Y_IN) || mxIsComplex(Y_IN) ||
        (MAX(m,n) != 4) || (MIN(m,n) != 1)) {
        mexErrMsgTxt("YPRIME requires that Y be a 4 x 1 vector.");
    }

    /* Create a matrix for the return argument */
    YP_OUT = mxCreateDoubleMatrix(m, n, mxREAL);

    /* Assign pointers to the various parameters */
    yp = mxGetPr(YP_OUT);

    t = mxGetPr(T_IN);
    y = mxGetPr(Y_IN);

```

```

    /* Do the actual computations in a subroutine */
    yprime(yp,t,y);
    return;
}

```

At the MATLAB prompt, type:

```
mex yprime.c
```

This uses the system compiler to create the MEX-file called yprime with the appropriate extension for your system.

You can now call yprime as if it were an M-function from within Matlab GUI command prompt.

```

Matlab prompt> yprime(1,1:4)
ans = 2.0000 8.9685 4.0000 -1.0947

```

To change the default compiler one can use mex –setup command and specify the specific options file.

How To Call Matlab Functions from C Programs:

Here we can use C/C++ as our Master program and invoke Matlab Engine routines for computations efficient in Matlab. We can map the candidate code down to FPGAs on the reconfigurable board and also exploit parallelism using MPI/PVM library routines. Some advantages of this approach are:

- One can invoke multiple sessions of Matlab on various nodes in a cluster of computers and call Matlab library routines, for example, to invert an array or to compute an FFT from your own program. When employed in this manner, MATLAB is a powerful and programmable mathematical subroutine library.
- Build an entire system for a specific task, for example, radar signature analysis or gas chromatography, where the front end (GUI) is programmed in C and the back end (analysis) is programmed in MATLAB, thereby shortening development time and getting higher performance by using a client server architectural approach and keeping the GUI on client machine running Matlab server on a separate node(s).

The MATLAB engine operates by running in the background as a separate process from your own program, which facilitates above approaches.

The following C program shows how to use Matlab Engine Functions:

```

/* This program invokes the Matlab Engine using engOpen function and then */
/* runs Matlab functions. In short it uses C as a master program which then*/

```

```

/* calls a Matlab function imacor.m which performs the operation of cross */
/* correlation of two images. */

#include <stdio.h>
#include "engine.h"
#define BUFSIZE 25000

main()
{
    Engine *ep;
    char buffer[BUFSIZE];
    int d;

    /****** starting matlab engine *****/

    ep=engOpen("\0"); /* start the Matlab Engine. */
    if (!ep) {
        fprintf(stderr, "\nCan't start MATLAB engine\n");
        return EXIT_FAILURE;
    } /* end if */

    engOutputBuffer(ep,buffer,25000);

    /* to store the Matlab outputs appearing on the Matlab GUI screen. */

    d=engEvalString(ep,"imacor"); /* calling Matlab function imacor.m */

    printf("Press Return to continue\n\n");
    fgetc(stdin);
    engClose(ep); /* close Matlab engine */
}

% File: imacor.m
% performs image correlation of 2 images 'image1.tiff' and 'image2.tiff'

function imacor()

% reading in 1-D data from files.
x1 = double(imread('image1.tiff','tif'));
x22 = double(imread('image2.tiff','tif'));

x2 = flipud(fliplr(x22));
sze1=size(x1);
sze2=size(x2);

r= sze1(1)+sze2(1)-1;
c= sze1(2)+sze2(2)-1;

```



```

% FFT of the data sets obtained from the images
x1f=fft2(x1,r,c);
x2f=fft2(x2,r,c);

xf=x1f.*x2f;

% Inverse FFT of the final data set to get the image data in the Time domain
x=ifft2(xf);

% plotting the output.
imagesc(abs(x));

# This Makefile compiles C programs which uses PVM libraries and also invokes
Matlab Engine. The Makefile will work only for SUN4SOL2 platform.

# Users will need to customize it in accordance to their programs. This is a
general Makefile which uses names file1.c and file2.c. The users will

# have to modify the file1 and file2 below and write their own file names in
their place. Be sure not to write them as yourfile.c and instead omit .c

# and just write yourfile. Please feel free to edit the file as you require and
suit your purpose.

# the clean part doesn't seem to work as yet. Need to look into it.

CC = cc

file1 = yourfile1

file2 = yourfile2

MATLAB = /mnt/sw/matlab6.1

INCLUDES = -I$(MATLAB)/extern/include -I$(MATLAB)/simulink/include -
DMATLAB_MEX_FILE -I$(MATLAB)/extern/include -I$(PVM_ROOT)/include -DSYSVBFUNC -
DSYSVSTR -DNOGETDTBLSIZ -DSYSVSIGNAL -DNOWAIT3 -DRSHCOMMAND=\"/usr/bin/rsh\"

CFLAGS = $(INCLUDES) -dalign -xlibmieee -Xc -D__EXTENSIONS__ -g

LIBDIR = /usr/local/pvm3/lib/$(PVM_ARCH)

LIBS = -L$(MATLAB)/extern/lib/sol2 -leng -lmx -lm -L$(PVM_ROOT)/lib/$(PVM_ARCH)
-lgpvm3 -lpvm3 -lnsl -lsocket $(LIBDIR)/libpvm3.a $(LIBDIR)/libgpvm3.a
$(LIBDIR)/libpvmtrc.a

$(file1) $(file2): $(file1).o $(file2).o mexversion.o
    $(CC) -g -o $(file2) $(file2).o mexversion.o $(LIBS)
    $(CC) -g -o $(file1) $(file1).o mexversion.o $(LIBS)
mexversion.o: mexversion.c
    $(CC) -c $(CFLAGS) mexversion.c
$(file2).o: $(file2).c
    $(CC) -c $(CFLAGS) $(file2).c
$(file1).o: $(file1).c
    $(CC) -c $(CFLAGS) $(file1).c
clean:
    /usr/bin/rm -f *.o

```